CHALMERS | GÖTEBORG UNIVERSITY

Space-Safe Transformations and Usage Analysis for Call-by-Need Languages

Jörgen Gustavsson

Department of Computing Science
CHALMERS UNIVERSITY OF TECHNOLOGY
GÖTEBORG UNIVERSITY
Göteborg Sweden 2001

Thesis for the Degree of Doctor of Philosophy

Space-Safe Transformations and Usage Analysis for Call-by-Need Languages

Jörgen Gustavsson

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Göteborg, May 2001

Space-Safe Transformations and Usage Analysis for Call-by-Need Languages
Jörgen Gustavsson
ISBN 91-628-4866-6

Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Printed at Chalmers, Göteborg, 2001

Abstract

This thesis is concerned with the resource consumption of lazy functional languages. It touches upon two aspects: how to reason about the space-safety of program transformations, and how to apply usage analysis for compiler optimisation. The thesis is a collection of articles.

In the first paper we study the notion of *space improvement*. We say that that a program fragment is *space improved* by another if and only if when we replace the former by the latter in any whole program the space behaviour is improved. We will refer to the induced equivalence as *space equivalence*.

We show that many of the extensional equivalences that lazy functional languages enjoy carry over as space equivalences, and we demonstrate that the space improvement theory can be used to show space properties of some interesting small programs. We also show that many extensionally equivalent program fragments are (sometimes surprisingly) not space equivalent by giving examples of whole programs where the asymptotic space behaviour changes if one replaces a program fragment by the another extensionally equivalent one.

An example of a transformation that is not a space equivalence in general is the *inlining* of function calls, i.e., replacing a function call with a copy of the body of the function with the arguments substituted for the formal parameters. In the second paper of thesis we study a class of automatic methods called *usage analyses* which can infer that an argument to a function is used at most once, and show that usage analyses can be used to guarantee the work and space safety of inlining.

Another application of usage analysis is compiler optimisation. In particular usage analysis can be used to avoid unnecessary closure updates. In the third paper of the thesis we present a usage analysis for this purpose which also provides additional information which can be used to optimise the bookkeeping of updates by avoiding unnecessary *update marker checks*.

In the fourth paper of the thesis we present a context sensitive usage analysis based on bounded usage polymorphic types. To implement the analysis efficiently we introduce a new form of constraint and in the fifth paper we show how the new form of constraints can be solved. The techniques can be applied not only to usage analysis but also to similar analyses. As an example of such, we present a *flow analysis* with flow subtyping, flow polymorphism and flow-polymorphic recursion, and show how it can be implemented in $O(n^3)$ time where n is the size of the explicitly typed program.

Keywords: lazy functional languages, equational theory, improvement theory, garbage collection, space use, space-equivalence, space-safety, work-safety, inlining, program analysis, usage analysis, sharing analysis, context sensitive, constraint solving.

This thesis contains five papers. They are based on ideas and results described in the following publications and reports:

1. *A Type Based Sharing Analysis for Update Avoidance and Optimisation.* In the Proceedings of the International Conference on Functional Programming, September, 1998, ACM Press.
2. *A Type Based Sharing Analysis for Update Avoidance and Optimisation.* Licentiate thesis, Göteborg University, May, 1999.
3. *A Foundation for Space-Safe Transformations of Call-by-Need Programs.* With David Sands. In the Proceedings of the Workshop on Higher Order Operational Techniques in Semantics, September 1999, volume 26 of Electronic Notes in Theoretical Computer Science. Elsevier, 1999.
4. *A Usage Analysis with Bounded Polymorphism and Subtyping.* With Josef Svenningsson. In the Proceedings of the Workshop on Implementation of Functional Languages, September 2000, Lecture Notes in Computer Science, volume 2011, Springer Verlag.
5. *Constraint Abstractions.* With Josef Svenningsson. In the the Proceedings of Symposium on Programs as Data Objects II, May 2001. To appear in Lecture Notes in Computer Science, volume 2053, Springer Verlag.
6. *Possibilities and Limitations of Call-by-Need Space Improvement.* With David Sands. To appear in the Proceedings of the International Conference on Functional Programming, September, 2001, ACM Press.

The first paper in this thesis unifies and expands on 3 and 6. The second paper is based on material in 3. The third paper is a reprint of 2 which is based on 1. The forth paper is a reprint of 4 and the fifth paper is a reprint of 5.

Contents

Acknowledgements

First of all I would like to thank my great supervisor David Sands. You have always taken the time to discuss with me whenever I have wanted to, even if it happened to be on a late Friday afternoon. Without your bright mind, good advice and encouragement I wouldn't have been able to finish this thesis. Thanks also for sometimes putting a little bit of pressure on me. I definitely needed that occasionally.

I would also like to thank John Hughes who was my supervisor during the first years of my PhD studies. John was also the one who got me into functional programming and who inspired me to apply for a PhD position – a choice I am very happy I made. Thanks John!

I would also like to thank Josef Svenningsson, my coauthor on the forth and fifth paper in this thesis. It has been very inspiring to work with you.

Thanks also to Johan Agat, Koen Claessen, Karl-Filip Faxén, Andrew Moran, Simon Peyton Jones, Jakob Rehof, Andrei Sabelfeld, Peter Sestoft, Josef Svenningsson, Jan Smith, Makoto Takeyama and Keith Wansbrough for discussions and feedback on the material in the thesis.

Working at the department of computing science has been great fun and I would like to thank everybody for the good time I have had here. Special thanks to Johan Agat, Ana Bove and Koen Claessen.

I would also like to thank my family and friends for contributing to my life outside the department. Finally, I would like to thank Birgit Grohe. I am very grateful for your love and support.

An overview of the thesis

This thesis is a collection of papers, each of them a self contained document. This part provides an overview of the papers and tries to present the contribution of the thesis in a manner accessible to a reader with a general background in computer science. It also provides connections between the different papers in the thesis.

The thesis is concerned with *lazy functional programming languages*, a class of programming languages which provides powerful mechanisms for abstraction making programs small and succinct. We will make no attempt to explain and argue the benefits of functional languages in this thesis, but take them for granted. In fact parts of the thesis can be seen as a critique of functional languages. For a reader who is interested in why functional programming should matter, Hughes article is warmly recommended [Hug89]. In the remainder of this overview we assume no prior knowledge of functional languages, but for the rest of the thesis some basic knowledge of functional programming is a prerequisite. A good introductory textbook for a reader with a general background in computer science is [Bir98].

It is often said that an advantage of lazy functional languages is that it is (relatively) easy to reason about functional programs – both formally and informally. One reason is that (purely) functional languages are free of *side effects*: if you call a function it is guaranteed that the function has no effects besides returning a result. It can neither manipulate global variables nor perform any input or output such as writing to the hard disk.

A consequence is that functional languages enjoys rich *equational theories*. An equational theory is a notion of equivalence between program fragments. Equational theories can be defined in numerous different ways but they all satisfy the intuitive property that if two program fragments are equivalent one may replace one with the other in any whole program without changing the outcome of running the program.

An equational theory provides an excellent tool for reasoning about programs because it allows a program to be step-wised *transformed* by replacing fragments of the program with other equivalent program fragments. In this way it is often possible to show that a (presumably) more efficient but complicated program is equivalent to an inefficient but simple one. Some people advocate it as a good programming methodology to first write simple and inefficient programs and then (semi-automatically) derive more efficient ones [PS83, PP96].

Even if only few programmers actually reason about their programs formally

or derive them, some knowledge of the equational theory can be very helpful in an informal argument for why a program works.

Reasoning about space

It is also often said about lazy functional languages that it is notoriously difficult to reason formally and informally about the time and space consumption of a program. One reason is that time and space consumption can be thought of as a kind of side effect: besides returning a result, a function uses up time and space. Another reason is that in a lazy functional language, arguments to functions are evaluated (i.e., computed) only when needed. So in a function call $f\ M$, the computation of the argument M is performed only if, and when, f requires the value of its argument. To reason about the time it takes to evaluate a function call it is of course crucial to know *whether* the argument is computed or not. To reason about memory consumption it is even more subtle because it then also matters *when* the argument is evaluated. For example, if a computation is postponed a large data structure may have to be kept in memory for a long time. An additional complication is that most implementations of lazy languages also ensure that if an argument is computed it is not subsequently recomputed should its value be required again. We will refer to languages implemented in this way as *call-by-need* languages.

When programmers are required to reason about the efficiency of their programs, the traditional equational theories fall short since they are concerned only with the outcome of programs and not with their time and memory consumption. If we replace a program fragment by another equivalent one it may radically change the time and space behaviour of the program. One says that the traditional equational theories are only concerned with the *extensional* behaviour of programs but ignores their *intentional* behaviour. Thus the traditional equational theories cannot be used for proving efficiency properties and they provide no intuition which aids informal reasoning about time and space consumption.

Recently, Moran and Sands defined and investigated an equational theory for a call-by-need language which takes the time consumption of programs into account [MS99]. In the first paper of this thesis we take the next step and investigate an equational theory of a call-by-need language which takes the memory usage of programs into account. The goal of the study was to develop techniques for formal reasoning about space usage but we also hope that our work may increase the general understanding of space usage and aid informal reasoning.

To reason formally about space use we need to precisely define how much memory a program uses up when it is executed. We base our definition on an *operational semantics* which is a precise mathematical definition which specifies in a step-wise manner how a program should be executed. In particular, our definition specifies how much memory is allocated in each execution step, and when it can be reclaimed and reused. In the remainder of this overview we assume no prior knowledge of operational semantics but for the rest of the thesis,

knowledge of operational semantics is assumed. An introduction to operational semantics can be found in [Win93].

With a space-aware operational semantics at hand, we can define two program fragments to be *space equivalent* if and only if we may replace one with the other in any whole program without changing the *asymptotic amount of space* used by the program. We are interested in the asymptotic amount of space rather than the absolute amount for the same reason one is usually interested in the asymptotic computational complexity of an algorithm rather than the absolute number of computation steps. We also define a corresponding asymmetric notion and say that that a program fragment is *space improved* by another if and only if when we replace the former by the latter in any whole program the space behaviour is improved.

An equational theory defined in this way is called an *operational theory* because the notion of equivalence is defined in terms of the operational semantics rather than in an axiomatic way by giving rules that specify when two programs fragments should be considered equivalent. Operational theories have become increasingly popular and the work in this thesis relies on techniques that have been developed quite recently. A good survey of these techniques can be found in [GP98].

Having defined a notion of space equivalence in this way, the question is whether there are any interesting space equivalences. Are any of the equivalences that functional languages enjoy also space equivalences? Is the equational theory rich enough to be useful? In the first paper in this thesis we show that many of the normal equivalences do carry over as space equivalences and we demonstrate that the equational theory can be used to show space properties of some interesting (but small) programs. We also show that many extensionally equivalent program fragments are (sometimes surprisingly) not space equivalent by giving examples of whole programs where the asymptotic space behaviour changes if one replaces a program fragment by the another extensionally equivalent one.

Usage analyses for work and space safe inlining

The work by Moran and Sands on reasoning about time and the work in this thesis on reasoning about space show that there are many interesting time and space equivalences. But some of the extensional equivalences do not carry over to the time and space aware theories. An example, and the subject of the second paper in the thesis, is the *inlining* of function calls, i.e., replacing a function call with a copy of the body of the function with the arguments substituted for the formal parameters. For example, inlining a call $square\ M$ to the squaring function yields $M * M$ (assuming that the body of $square\ x$ is $x * x$) and it is an example of a transformation which can change the asymptotic time and space behaviour of some programs. The intuitive explanation of why this may happen (the second paper of this thesis provides a more detailed and technical explanation) is that in $M * M$, we evaluate M twice but call-by-need ensures that when evaluating $square\ M$ the argument M is evaluated only once.

But inlining does not always change the time and space behaviour of a program. Whether it does, depends on the function that is called, which arguments are passed and how the result of the function is used. In short it may depend on the entire program. Thus to show that inlining is *work* and *space safe* (i.e., preserves the time and space behaviour of the program) may involve reasoning about the entire program rather than reasoning locally about the program fragment that is modified.

Turner, Wadler and Mossin [TWM95] proposed to use *usage analyses* (sometimes also called *sharing analyses*) to decide if inlining is work-safe. Usage analyses are automatic techniques for deciding when an argument is used at most once and the idea is that it should be safe to inline a function call if the arguments are used at most once. Whether an argument is used at most once may depend on the entire program, so usage analyses are inherently global. The more recent usage analyses have been proved correct in the sense that when the analysis claims that an argument is used at most once then it is indeed the case. But despite the fact that Turner et al discuss inlining in some detail, as far as we are aware, it has remained an open problem to actually prove that any of the usage analyses in the litterature guarantee work-safety. Another question (one which to our knowledge has not even been posed) is whether usage analyses might also guarantee space safety.

In the second paper of this thesis, rather than showing work and space safety for any particular usage analysis, we pose the question: can the intuitive semantic criteria "used at most once" guarantee work and space safety? The paper provides the answer that with a slight strengthening (see the second paper in the thesis for the details) the used-at-most-once criteria can indeed guarantee both work and space safety.

Usage analysis for complier back-end optimisation

The topic of the third paper in the thesis is a usage analysis, i.e., an automatic method for determining how many times the argument to a function is used. Although it can be used to guarantee work and space-safe inlining, as described in the previous section, the focus of the third paper is another application of usage analysis: optimisations in a compiler back-end. [1]

To understand how usage analysis can be used to optimise the back-end of a compiler it is necessary to know a little bit about how call-by-need is implemented. Recall that, in call-by-need, arguments to functions are evaluated only if needed and at most once. It is implemented as follows. When a function call $f\ M$ is executed, a *representation* of the expression M (and the parts of the environment to which it refers) is built in memory. This representation is called a *closure* for M. After the closure has been built, the function f is called with a pointer to the closure as the argument. When (if at all) the value of

[1] Chronological note: The reason for why the third paper doesn't mention inlining is that it is a reprint of the author's licentiate thesis which was written before the second paper, and at that time it was not clear to us whether the analysis could actually guarantee work and space-safe inlining.

M is required the expression represented by the closure is evaluated. After the evaluation has finished the closure is *updated* (i.e., overwritten) with the result of the evaluation, so that if the value of M is required again it need not to be reevaluated. But if the value of M is not needed again the update of the closure is unnecessary. Usage analysis can be used to avoid these unnecessary updates by statically (i.e., at compile time) determining that the value of a closure will be required at most once.

Besides the cost of performing updates there is also a cost associated with the bookkeeping of updates, that is keeping track of when and where to update. The usage analysis in the third paper also provides information which allows the bookkeeping cost of the update machinery to be reduced. To understand this aspect of the analysis it is necessary to know details of how the bookkeeping machinery is implemented and we refer the interested reader to the third paper of this thesis.

It is important to note that our usage analysis can not always say that an argument is used is used at most once even if it is actually the case. It happens that the analysis comes to the conclusion that it "doesn't know". The reason is that it is an undecidable property and thus any automatic method must approximate and sometimes yield the "don't know" answer. Automatic methods for inferring properties of program are often called *program analyses* or *static analyses*. In the third paper we assume some basic knowledge in the area of program analysis, especially type based program analysis. An thorough introduction to this topics can be found in [NNH99].

Context sensitive usage analysis

A weakness of the usage analysis presented in the third paper is that it is not context sensitive in the following sense. If a function is called in several different places then the nature of the calls may be different. For example the results of the function calls may be used differently. But the analysis in the third paper lumps these calls together as if they were one. As a result the analysis answers with "don't know" more often than one might hope. As programs get larger it becomes more and more common that a function is called at numerous places. As a result the precision of the analysis degrades as programs grow. This problem is the subject of the forth and the fifth paper of the thesis.

In the forth paper we define a usage analysis which is context sensitive and therefore is more precise than the analysis in the third paper (but it does not provide information for optimising the bookkeeping of updates).

Context sensitive analyses provides more accurate answers so it might not come as a surprise that they are also more computationally expensive. The subject of the fifth paper is how the usage analysis in the forth paper can be implemented efficiently. It is a separate paper because the techniques applies not only to usage analysis but also to other similar analyses. As an example of such a similar analysis, we present a *flow analysis* which is a program analysis that (for example) tries to predict where a value computed at a certain point in the program is used.

References

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, 1998.

[GP98] A. D. Gordon and A. M. Pitts, editors. *Higher Order Operational Techniques in Semantics.* Cambridge University Press, 1998.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[MS99] Andrew Moran and David Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Proc. POPL'99*, pages 43–56. ACM Press, January 1999.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer Verlag, 1999.

[PP96] A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[PS83] H. Partsch and R. Steinbruggen. Program Transformation Systems. *ACM Computing Surveys*, 15(3):199–236, 1983.

[TWM95] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proc. FPCA'95*, pages 1–11. ACM Press, June 1995.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: an Introduction.* MIT Press, 1993.

Paper I

Space Safe Transformations of Call-by-Need Programs

Space Safe Transformations of Call-by-Need Programs

Jörgen Gustavsson David Sands

Abstract

We introduce a space-improvement relation on programs which guarantees that whenever M is improved by N, replacement of M by N in a program can never lead to asymptotically worse space (heap or stack) behaviour, for a particular model of garbage collection. This study takes place in the context of a call-by-need programming language. For languages implemented using call-by-need, e.g, Haskell, space behaviour is notoriously difficult to predict and analyse, and even innocent-looking equivalences like $x + y = y + x$ can change the asymptotic space requirements of some programs. Despite this, we establish a fairly rich collection of improvement laws, with the help of a context lemma for a finer-grained improvement relation, *strong space improvement*. We also show that the asymptotic space improvement relation is semantically badly behaved, but that the theory of strong space improvement possesses a fixed-point induction theorem which permits the derivation of improvement properties for recursive definitions. With the help of this tool we explore the landscape of space improvement by considering a range of classical program transformations seeking the answers to the following questions: is the improvement relation inhabited by interesting program transformations? and, if so, how might they be established? [1].

1 Introduction

The space-usage of lazy functional programs is perhaps the most thorny problem facing programmers using languages such as Haskell. Almost all programmers unable to predict or control the space behaviour of their lazy programs. Even the most advanced programmers, who are able to visualise the space use of their programs, complain that the "state-of-the-art" compilers *introduce* space-leaks into programs that they believe ought to be space-efficient.

In recent years a successful line of research into profiling tools for lazy functional languages [RW93, RR96] has greatly improved a programmer's chances of locating sources of space leaks. But apart from a few high-level operational semantics which claim to model space behaviour, to the best of our knowledge

[1] This paper subsumes its predecessors [GS99, GS01]

there have been no formal/theoretical/semantics-based approaches to reasoning about space behaviour of programs.

Rather than tackling the problem of determining the absolute space behaviour of a program, in this paper we study notions of relative space efficiency. We pose the question: when is it space-safe to replace one program fragment by another? To this end we introduce a space-*improvement* relation on terms, which guarantees that whenever M is improved by N, replacement of M by N in a program can never lead to asymptotically worse space (heap or stack) behaviour, for a particular model of computation and garbage collection.

The fact that we only aim to prevent *asymptotic* worsening might seem rather weak. One reason is that we (wish to) work with high-level semantic models of space behaviour, so it is not meaningful for us to make stronger claims. Another reason is that asymptotic changes in space behaviour are not at all unusual. (We consider such an example below.)

Why is the space behaviour of lazy functional programs difficult to predict? One reason is of course that all memory management is automatic, coupled with the fact that the heap allocation rate of functional programs is very high; just about everything lives in the heap. A second reason is that the non-strict evaluation order that is required by the language specification means that computation-order bears no obvious relation to textual structure of code. The third, and perhaps most subtle reason is that all realistic implementations of lazy languages use call-by-need. Call-by-need optimises call-by-name by ensuring that when evaluating a given function application, arguments are evaluated at most once. The effect of sharing is to reduce – often dramatically – the time required to execute a program. But the effect of this additional sharing on the space behaviour is to prolong the lifetime of data, and this is often at the cost of space.

As an illustration of some of these problems, consider one of the most innocent of the extensional equivalences that functional programming languages enjoy: $x + y = y + x$. In a lazy functional language the transformation is not *space safe*; there are programs for which this innocent-looking transformation will change their space complexity. Now consider the following family of Haskell programs, indexed by some integer n:

$$\begin{array}{l}\mathsf{let}\ xs = [1..n]\ ;\ x = \mathit{head}\ xs\ ;\ y = \mathit{last}\ xs \\ \mathsf{in}\ x + y\end{array}$$

If addition is evaluated from left-to-right then this program runs in constant space. First x is evaluated to obtain 1, then y is evaluated, which involves constructing and traversing the entire list $[1..n]$. Fortunately, the combination of lazy evaluation, tail recursion and garbage collection guarantees that as this list is constructed and traversed it can also be garbage collected, and thus the computation requires only constant space. But if $x + y$ is replaced by $y + x$ the *space* required is $\mathcal{O}(n)$. This is because when y builds and traverses the list $[1..n]$, the elements cannot be garbage-collected because the whole list is still

live via the occurrence of xs in the body of x. So we can conclude that replacing $x + y$ by $y + x$ can give an asymptotic change in space behaviour – i.e., there is no constant which bounds the potential worsening in space when this law is applied in an arbitrary context. So our theory of improvement will not relate this particular pair of terms.

Expressions that fall outside our improvement theory are easy to find (see e.g., [PJ87] for more tricky examples), but given the example above it is not immediately clear that there are *any* interesting transformations which are space improvements. This paper seeks an answer to the following questions: is the improvement relation inhabited by interesting program transformations, and, if so, how might they be established? For example, is the associativity property of list concatenation a space improvement in either direction? Are typical tail recursion optimisations space safe? In this article we show that there are indeed many valid basic space-improvement laws. For example, the *beta-var* transformation between $(\lambda x.M)\, y$ and $M[{}^{y}/_{x}]$ is shown to be a space improvement if y occurs in $M[{}^{y}/_{x}]$. But basic laws are not enough. With the basic laws alone it is not possible to show any improvements beyond those obtainable by composing the basic laws. To reason about recursive definitions we provide a fixed-point induction theorem for a finer-grained relation *strong space improvement*. The reason to introduce the finer-grained notion is that the asymptotic space improvement relation, is semantically badly behaved, it is discontinuous with respect to finite unfoldings.

With the help of this tool we explore the landscape of space improvement by considering a range of classical program transformations, and uncovering a number of fundamental limitations to what can be achieved by local improvement.

Overview The remainder of the article is organised as follows. **Section 2** gives the syntax and operational semantics of our language. **Section 3** defines what we mean by the space-use of programs, in terms of a definition of garbage collection for abstract-machine configurations. We informally argue the ways in which this definition agrees with lower-level models, and mention a number of subtle choices and variations in actual implementation methods. **Section 4** defines the main improvement relation, *weak improvement*, and presents the basic properties of this relation. **Section 5** describes a finer-grained improvement relation, *strong improvement*, and establishes a context lemma and a fixed-point induction principle. **Section 6** applies the theory to investigate a range of transformations. **Section 7** gives the proof of the context lemma. **Section 8** gives the proofs of some selected basic laws. **Section 9** describes related work. **Section 10** concludes and proposes future work. **Appendix A** and **Appendix B** considers two language extensions: unboxed integers and pattern-bindings.

2 Operational Semantics

Our language is an untyped lambda calculus with recursive lets, structured data, case expressions, a strictness combinator, bounded integers (ranged over by n and m) with addition and a zero test. In appendix B we extend the language with patterns bindings in let expressions. We work with a restricted syntax in which arguments to functions (including constructors) are always variables:

$$\begin{array}{rcl} L, M, N & ::= & x \mid \lambda x.M \mid M\,x \mid c\,\vec{x} \mid \mathsf{seq}\,M\,N \\ & \mid & n \mid M + N \mid \mathsf{add}_n M \mid \mathsf{iszero}\,M \\ & \mid & \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N \mid \mathsf{case}\ M\ \mathsf{of}\ \{c_i\,\vec{x}_i \to N_i\} \end{array}$$

The syntactic restriction is now rather standard, following its use in core language of the Glasgow Haskell compiler, *e.g.*, [PJPS96, PJS98], and in [Lau93, Ses97]. In examples we will sometimes use unrestricted application $M\,N$ as syntactic sugar for $\mathsf{let}\ \{x = N\}\ \mathsf{in}\ M\,x$ where x is a fresh variable. Similarly for constructor expressions.

All constructors have a fixed arity, and are assumed to be saturated. By $c\,\vec{x}$ we mean $c\,x_1\,\cdots\,x_n$. The only values are lambda expressions, fully-applied constructors and integers. Throughout, x, y, z etc., will range over variables, c over constructor names, and V and W over values ($\lambda x.M \mid c\,\vec{x} \mid n$). We will write

$$\mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N$$

as a shorthand for $\mathsf{let}\ \{x_1 = M_1, \ldots, x_n = M_n\}\ \mathsf{in}\ N$ where the $\vec{x}$ are distinct, the order of bindings is not syntactically significant, and the $\vec{x}$ are considered bound in N *and* the $\vec{M}$ (so our lets are recursive). We will use Γ to range over a set of such distinct bindings. Similarly we write

$$\mathsf{case}\ M\ \mathsf{of}\ \{c_i\,\vec{x}_i \to N_i\}$$

for

$$\mathsf{case}\ M\ \mathsf{of}\ \{c_1\,\vec{x}_1 \to N_1 \mid \cdots \mid c_m\,\vec{x}_m \to N_m\}$$

where each $\vec{x}_i$ is a vector of distinct variables, and the c_i are distinct constructors. In addition, we will sometimes write *alts* as an abbreviation for case alternatives $\{c_i\,\vec{x}_i \to N_i\}$.

Our integers are bounded (i.e., for an integer n, $MININT \leq n \leq MAXINT$) so that they can be represented in constant space. For simplicity, no exception occurs at overflow. Instead the result wraps as in e.g., C. The functions add_n are included for convenience in the definition of the abstract machine, and represent an intermediate step in the addition of n to a term.

We have included a strictness combinator $\mathsf{seq}\,M\,N$ which first evalaluates M, throws away the result and then continues with N. The strictness combinator

is necessary to define space efficient versions of some functions, such as the *sum* function with a strict accumulator (see Section 6).

The only kind of substitution that we consider is *variable for variable*, with σ ranging over such substitutions. The simultaneous substitution of one vector of variables for another will be written $M[\vec{y}/\vec{x}]$, where the $\vec{x}$ are assumed to be distinct (but the $\vec{y}$ need not be).

2.1 The Abstract Machine

The semantics presented in this section is essentially Sestoft's "mark 1" abstract machine for laziness [Ses97]. Transitions are over configurations consisting of a *heap*, containing bindings, the expression currently being evaluated, and a *stack*. We write $\langle\, \Gamma,\ M,\ S \,\rangle$ for the abstract machine configuration with heap Γ, expression M, and stack S and we will use Σ and Φ to range over such configurations. A heap is a set of bindings; we denote the empty heap by $\emptyset$, and the addition of a group of fresh bindings $\vec{x} = \vec{M}$ to a heap Γ by juxtaposition: $\Gamma\{\vec{x} = \vec{M}\}$. The stack written $b : S$ will denote the stack S with b pushed on the top. The empty stack is denoted by ϵ.

Stack elements are either:

- a *reduction context*, or
- an *update marker* $\#x$, indicating that the result of the current computation should be bound to the variable x in the heap.

The reduction contexts on the stack are shallow contexts containing a single hole in a "reduction" position - i.e. in a position where the current computation is being performed. They are defined as:

$$\begin{aligned} R ::= & [\cdot]\, x \mid \mathsf{case}\ [\cdot]\ \mathsf{of}\ \{c_i\, \vec{x}_i \to N_i\} \mid \mathsf{seq}\, [\cdot]\, M \mid \\ & [\cdot] + M \mid \mathsf{add}_n [\cdot] \mid \mathsf{iszero}\, [\cdot] \end{aligned}$$

We will refer to the set of variables bound by Γ as $\mathrm{dom}\,\Gamma$, and to the set of variables marked for update in a stack S as $\mathrm{dom}\, S$. Update markers should be thought of as binding occurrences of variables. A configuration is *well-formed* if $\mathrm{dom}\,\Gamma$ and $\mathrm{dom}\, S$ are disjoint. We write $\mathrm{dom}(\Gamma, S)$ for their union. For a configuration $\langle\, \Gamma,\ M,\ S \,\rangle$ to be closed, any free variables in Γ, M, and S must be contained in $\mathrm{dom}(\Gamma, S)$. The free variables of a term M will be denoted $\mathsf{FV}(M)$; for a vector of terms $\vec{M}$, we will write $\mathsf{FV}(\vec{M})$.

The abstract machine semantics is presented in Figure 1; we implicitly restrict the definition to well-formed closed configurations.

The first group of rules are the standard call-by-need rules. Rules (*Lookup*) and (*Update*) concern evaluation of variables. To begin evaluation of x, we remove the binding $x = M$ from the heap and start evaluating M, with x, marked for update, pushed onto the stack. Rule (*Update*) applies when this

$$\langle \Gamma\{x = M\},\ x,\ S \rangle \to \langle \Gamma,\ M,\ \#x : S \rangle \qquad (\mathit{Lookup})$$

$$\langle \Gamma,\ V,\ \#x : S \rangle \to \langle \Gamma\{x = V\},\ V,\ S \rangle \qquad (\mathit{Update})$$

$$\langle \Gamma,\ \mathsf{let}\ \Gamma'\ \mathsf{in}\ N,\ S \rangle \to \langle \Gamma\Gamma',\ N,\ S \rangle \qquad (\mathit{Letrec})$$

$$\langle \Gamma,\ R[M],\ S \rangle \to \langle \Gamma,\ M,\ R : S \rangle \qquad (\mathit{Push})$$

$$\langle \Gamma,\ V,\ R : S \rangle \to \langle \Gamma,\ M,\ S \rangle \quad \text{if } R[V] \rightsquigarrow M \qquad (\mathit{Reduce})$$

$$\begin{aligned}
(\lambda x.M)\, y &\rightsquigarrow M\,[{}^{y}/_{x}] \\
\mathsf{case}\ c_j\, \vec{y}\ \mathsf{of}\ \{c_i\, \vec{x}_i \to M_i\} &\rightsquigarrow M_j[{}^{\vec{y}}/_{\vec{x}_j}] \\
\mathsf{seq}\, V\, M &\rightsquigarrow M \\
m + N &\rightsquigarrow \mathsf{add}_m N \\
\mathsf{add}_m n &\rightsquigarrow \ulcorner m + n \urcorner \\
\mathsf{iszero}\, m &\rightsquigarrow \begin{cases} \mathsf{true} & \text{if } m = 0 \\ \mathsf{false} & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 1: Abstract machine semantics

evaluation is finished, and we may update the heap with the new binding for x. Rule (*Letrec*) adds a set of bindings to the heap.

The basic computation rules are captured by the (*Push*) and (*Reduce*) rules schemas. The rule (*Push*) allows us to get to the heart of the evaluation by "unwinding" a shallow reduction context. When the term to be evaluated is a value and there is a reduction context on the stack, the (*Reduce*) rule is applied.

3 Space Use and Garbage Collection

A desired property of our model of space-use is that it is true to actual implementations. Unfortunately, different abstract machines and garbage collection strategies differ in their asymptotic space behaviour. Consider for example an application of the function

$$f = \lambda x.\mathsf{let}\ y = f\ y\ \mathsf{in}\ y$$

using some of the main Haskell implementations.[2] This runs in constant space under HUGS'98 and GHC 4.01, but runs out of stack in hbc 0.9999.5a, and in some older versions of GHC. Later in this section we will try to explain the differences.

[2] www.haskell.org/implementations.html

Given the different space behaviours of different implementations there is no hope that we can construct a theory which applies to all implementations. Although we will choose a particular model of space use we believe that most of the results and techniques developed in this paper can be adapted to any reasonable model. Later in this section we discuss some of the subtle ways in which different abstract machines and implementations described in the literature differ from our model and each other. Bakewell and Runciman [BR00a] focus on techniques for comparing different evaluators.

Another point of dispute is whether to distinguish between heap and stack space. Many implementations allocate separate memory for the heap and the stack, but in principle the stack and the heap can share the same memory. So, should a transformation which trades heap for stack, or vice versa, be rejected? And do such transformation show up "in practice"? We focus mainly on a theory which keeps stack and heap usage separate. However, we will see examples of transformations which usefully trade stack for heap.

3.1 Measuring space

We measure the heap space occupied by a configuration by counting the number of bindings in the heap and the number of update markers on the stack. We count update markers on the stack as also occupying heap space, since in a typical implementation an update marker refers to a so-called "blackhole closure" in the heap – a placeholder where the update eventually will take place. We will count every binding as occupying one unit of space.

In practice the size of a binding varies since a binding is typically represented by a tag or a code pointer plus an environment with one entry for every free variable. However, the right hand side of every binding is a (possibly renamed) subexpression of the original program, (a property of the semantics sometimes called *semi-compositionality*) so counting it as occupying one unit of space gives a measure which is within a constant factor (depending only on the program size) of the actual space used. Integers are an exception to this claim, but recall that our integers are bounded so they can also be represented in a constant amount of space.

We measure stack space by simply counting the number of elements on the stack, so an update marker will be viewed as occupying both heap and stack space. In practice every element on the stack does not occupy the same amount of space, but again, semi-compositionality of the abstract machine assures that our measure is within a program-size-dependent constant factor. The size of a configuration, written $|\Sigma|$ is a pair (h, s) where h and s is the amount of heap and stack respectively occupied by the the configuration.

3.2 Garbage collection

We cannot reason about space usage without modelling garbage collection. During a computation, garbage collection allows us to decrease the amount of space used by a configuration. It is modelled simply by the removal of any number of bindings and update markers from the heap and the stack respectively, *providing that the configuration remains closed.*

Definition 3.1 (GC)
Garbage collection can be applied to a closed configuration Σ to obtain Σ', written $\Sigma \gg \Sigma'$ if and only if Σ' is closed, and can be obtained from Σ by removing zero or more bindings and update markers from the heap and the stack respectively.

This is an accessibility-based definition as found in e.g., the gc-reduction rule of [MH98]. The removal of update-markers from the stack is not surprising given that they are viewed as the binding occurrences of the variables in question.

We are now ready to define what it means for a computation to be possible in certain fixed amount of space.

Definition 3.2 (Convergence in fixed space)

1. $\Sigma \rightarrow_{(h,s)} \Sigma' \stackrel{\text{def}}{=} \Sigma \rightarrow \Sigma'$ and $|\Sigma| \leq (h, s)$.

2. $\twoheadrightarrow_{(h,s)} \stackrel{\text{def}}{=}$ the reflexive and transitive closure of the relational composition of $\rightarrow_{(h,s)}$ and $\gg$.

3. $\Sigma\Downarrow_{(h,s)} \stackrel{\text{def}}{=} \exists \Delta, V.\Sigma \twoheadrightarrow_{(h,s)} \langle\, \Delta,\ V,\ \epsilon \,\rangle$ and $|\langle\, \Delta,\ V,\ \epsilon \,\rangle| \leq (h, s)$.

4. $M\Downarrow_{(h,s)} \stackrel{\text{def}}{=} \langle\, \emptyset,\ M,\ \epsilon \,\rangle\Downarrow_{(h,s)}$.

We read $M\Downarrow_{(h,s)}$ as M can converge within (h, s) space, i.e., the maximum heap, and stack is less than or equal to h and s respectively. Note that, with this definition, if a binding is garbage collected immediately after it has been allocated it does not account for any space. In real implementations the binding would of course momentarily take up one unit of space even if it is garbage collected immediately. However, our model is within a constant factor.

3.3 Some subtleties

Different implementations vary in their space behaviour in a number of rather subtle ways. We will discuss some of those points below and how they relate to our particular space model.

Environment trimming Our abstract machine is based on substitution of variables for variables, but lower level abstract machines are usually based on environments [Ses97]. To avoid space leaks in environment-based machines it is crucial to remove redundant bindings from environments on the stack. This is sometimes called *environment trimming* [Ses97] or *stack stubbing* [PJ92]. Some implementations do not properly trim environments and programs like

$$f\,x\,y = \mathsf{case}\ g\,x\ \mathsf{of}\ \mathit{alts},$$

where there is no occurence of x in *alts*, can lead to space leaks because a reference to x is kept on the stack during the evaluation of $g\,x$ [Röj95]. Our definition of space is consistent with an environment machine which *does* perform environment trimming.

Blackholing In our abstract machine, the lookup rule removes the binding from the heap while it is being evaluated. This corresponds to so called "black holing" in real implementations where the closure is overwritten with a special "black hole closure" without free variables [PJ92]. In some early implementations the closure was instead left untouched in the heap [RW93, Jon92]. This has the effect that the garbage collector can not reclaim space that the free variables of the closure hangs on to.

Garbage collection of update markers In our model we allow for the garbage collection of update markers which allows our example from the beginning of this section, an application of f

$$f = \lambda x.\mathsf{let}\ y = f\ y\ \mathsf{in}\ y,$$

to run in constant space – as it does in HUGS'98 and GHC 4.01, but not in hbc 0.9999.5a, or in some older versions of GHC. The collection of update markers could explain the different behaviours of the implementations but it can also be explained by the implementations *shortcutting update marker chains*. We explain this trick later in this section.

Avoiding value copying When running

$$\mathsf{let}\ x = 1 + 2, y = id\,x\ \mathsf{in}\ y + M$$

in our abstract machine we will end up with both x and y bound to 3 in the heap. Some implementations would instead bind x to 3 and create an indirection $y \mapsto x$ from y to x (or vice versa). If this is combined with a garbage collector which can shortcut indirections (by in this case replacing all occurrences of y with x and removing $y \mapsto x$) then space can be saved. This can only reduce the total space used by a constant factor, but it can have a quite dramatic effect in practice [RW93]. However since our space model is only adequate up

to constant factors anyway, this is not a serious drawback of our space model. For implementations that create indirections in this way it is important that the garbage collector can shortcut indirections. Otherwise not much would be gained, and in our example, the space for x cannot be reclaimed before $y \mapsto x$ is reclaimed, thus possibly increasing the space used (although we believe the additional space is within a constant factor). Since our abstract machine does not create indirections (other than those which occur textually in the program) we have not included shortcutting of indirections in our garbage collector.

Shortcutting update marker chains Sometimes two or more update markers, say $\#x$ and $\#y$, end up on top of each other on the stack. Then both x and y will eventually be updated with the same value or, in implementations which introduce indirections, one will be indirected to the other. One can preclude this situation by never pushing update markers on top of each other: if $\#x$ is already on the stack, an indirection $y \mapsto x$ is created instead of pushing $\#y$. When this is combined with garbage collection of indirections, the effect is similar to the combined effect of garbage collection of update markers, avoiding value copying and garbage collecting indirections. As far as we know this trick has not been documented, but a variation of it is used in the GHC compiler[3] where the garbage collector removes update markers pushed on top of each other by indirection one to the other.

Pattern bindings Patterns in let bindings like

$$\mathsf{let}\ (x, y) = M\ \mathsf{in}\ N$$

are an important feature of real lazy languages such as Haskell. They might be encoded in our language in the following manner

$$\mathsf{let}\ p = M, x = \mathsf{fst}\, p, y = \mathsf{snd}\, p\ \mathsf{in}\ N$$

This encoding seems to be used in, for example HUGS'98, but can lead to undesirable space behaviours for some functions [Hug83] (see Section 6, case study 6, for an example). Intriguingly, pattern bindings are linked to the intensional expressiveness of the language. Hughes has argued that it is impossible to define a certain function *split*, which splits the input into the first line and the rest, in a space efficient way using a particular lazy evaluator similar to ours [Hug83]. There have been several proposals to solve this problem due to Hughes [Hug83], Wadler [Wad87] and Sparud [Spa93]. We discuss their proposals in some detail in Section 6. Sparud's suggestion was to have pattern bindings as a first class construct which the evaluator treats in a space efficient manner. We have adopted Sparud's proposal as an extension to our language. The semantics is in Appendix B. With this extension it is possible to define *split* space efficiently.

[3] Simon Peyton Jones, Personal communication, June 1999.

4 Weak Space Improvement

In the previous section we defined a notion of space which we believe is realistic in the sense that an actual implementation (using our reasonably aggressive garbage collection) will require space within a constant factor of our abstract measure, where the constant depends on the size of the program to be executed.

In this section we define *space improvement within a constant factor* – what we will simply refer to as *Weak Improvement* – which says that if M is improved by N, replacing M by N in any program context will never lead to more than a constant factor worsening in space behaviour, where the constant factor is independent of the context.

The starting point for an operational theory is usually an approximation and an equivalence defined in terms of *program contexts*. Program contexts are usually introduced as "programs with holes", the intention being that an expression is to be "plugged into" all of the holes in the context. The central idea is that to compare the behaviour of two terms one should compare their behaviour in all program contexts.

We will use contexts such that holes may not occur in argument positions of an application or a constructor, for if this were the case, then filling a hole (with a non variable) would violate the syntax since it could yield a non-restricted application. Contexts may contain zero or more occurrences of the hole, and as usual the operation of filling a hole with a term can cause variables in the term to become captured. We will use $\mathbb{C}$ and $\mathbb{D}$ to range over contexts. The grammar of contexts is as follows.

$$\begin{array}{rcl} \mathbb{C},\mathbb{D} & ::= & [\cdot] \mid x \mid \lambda x.\mathbb{C} \mid \mathbb{C}\,x \mid c\,\vec{x} \mid \mathsf{seq}\,\mathbb{C}\,\mathbb{D} \\ & \mid & n \mid \mathbb{C}+\mathbb{D} \mid \mathsf{add}_n\mathbb{C} \mid \mathsf{iszero}\,\mathbb{C} \\ & \mid & \mathsf{let}\ \{\vec{x}=\vec{\mathbb{C}}\}\ \mathsf{in}\ \mathbb{D} \mid \mathsf{case}\ \mathbb{C}\ \mathsf{of}\ \{c_i\,\vec{x}_i \to \mathbb{D}_i\} \end{array}$$

Definition 4.1 (Weak Improvement)
We say that M *is weakly improved by* N, written $M \underset{\approx}{\trianglerighteq} N$, if there exists a linear function $f \in \mathbb{N} \to \mathbb{N}$ such that for all $\mathbb{C}$, σ such that $\mathbb{C}[M\sigma]$ and $\mathbb{C}[N\sigma]$ are closed,

$$\mathbb{C}[M\sigma]\Downarrow_{(h,s)} \implies \mathbb{C}[N\sigma]\Downarrow_{(f(h),f(s))}.$$

So $M \underset{\approx}{\trianglerighteq} N$ means that N never takes up more than a constant factor more space than M (but it might still use non-constant factor less space).

Note that we compare the behaviour of all *substitution instances* of the two terms in all program contexts rather than comparing the two terms in all program contexts. As a result weak improvement is closed under substitution by defintion – a very useful property. We believe that the two definitions are equivalent but we have failed to prove it.

We write $M \underset{\approx}{\triangleleft\triangleright} N$ to mean that $M \underset{\approx}{\trianglerighteq} N$ and $N \underset{\approx}{\trianglerighteq} M$.

Proposition 4.2 (Precongruence)
$\stackrel{\triangleright}{\approx}$ is a *precongruence* – i.e., it is a transitive and reflexive relation which is preserved by contexts.

PROOF. The proof of all but transitivity is immediate. Transitivity follows from the fact that the composition of any two linear functions is linear. □

4.1 Limitations of Weak Improvement

In this section we address some inherent limitations of weak improvement which serves as the motivation for studying a stronger notion of space improvement in the next section. As these are negative results we will not prove them in detail but we sketch the proofs since we think they provide important insight into the nature of weak improvement.

The Free Variable Restriction The first limitation is fundamental, and highlights the significance of free variables in this theory:

Theorem 4.3 (Free Variable Restriction)
If $\mathsf{FV}(M) \not\supseteq \mathsf{FV}(N)$ then $M \stackrel{\not\triangleright}{\approx} N$

PROOF. (Sketch) Suppose that $M \stackrel{\triangleright}{\approx} N$. Then there exits a linear function f which bounds the extra space required to compute with N instead of M. Assume, towards a contradiction, that there exists a variable x such that $x \in \mathsf{FV}(N)$ but $x \notin \mathsf{FV}(M)$. Without loss of generality we can assume that $\mathsf{FV}(N) = \{x\}$ and $\mathsf{FV}(M) = \emptyset$ (since by congruence of $\stackrel{\triangleright}{\approx}$ we can wrap a context around M and N which ensures this property). Now consider the context $\mathbb{C}$:

$$
\begin{array}{lll}
\mathsf{let} & \mathit{traverse} = \lambda xs.\mathsf{case}\ xs\ \mathsf{of} & \\
 & \qquad\qquad \mathsf{nil} \to 1 & \\
 & \qquad\qquad h : t \to \mathit{traverse}\, t & \\
 & \mathit{count} = \lambda n.\ \mathsf{case}\ \mathsf{iszero}\ n\ \mathsf{of} & \\
 & \qquad\qquad \mathsf{true} \to \mathsf{nil} & \\
 & \qquad\qquad \mathsf{false} \to \mathsf{let} & a = n - 1 \\
 & \qquad\qquad\qquad\qquad\quad & r = \mathit{count}\, a \\
 & \qquad\qquad\qquad\quad \mathsf{in} & n : r \\
 & x = \mathit{count}\, \mathbf{k} & \\
 & z = [\cdot] & \\
\mathsf{in} & \mathit{traverse}\, x + (\lambda y.1)z &
\end{array}
$$

where **:** is an infix cons constructor. Recall that + evaluates its arguments from left to right. It can be seen (we omit a formal proof, which would be somewhat tedious) that $\mathbb{C}[M]$ evaluates in constant space, independent of $\mathbf{k}$. This is because the list $\mathit{count}\,\mathbf{k}$ can be garbage collected as it is traversed. However $\mathbb{C}[N]$ requires space proportional to $\mathbf{k}$, since there is a (dead code)

reference to x which prevents any of the list from being collected until it has been completely constructed. Since we can make **k** arbitrarily large we cannot have $M \overset{\rhd}{\approx} N$. □

The sketch proof above relies on unbounded integers. A similar example can be constructed using just a finite set of constructors and a logarithmic-space encoding of **k**.

Failure of the Context Lemma A standard result for any operational theory is a *context lemma* [Mil77]. A context lemma in this case would establish that to prove that M is weakly improved by N, one only needs to compare their behaviour with respect to a much smaller set of contexts, namely the context which immediately need to evaluate their holes.

Despite our efforts we were not able to prove the context lemma. The reason is that the context lemma, as we envisage it, does not hold for weak improvement:

Theorem 4.4 (Failure of the context lemma)
There exist terms M and N with $\mathsf{FV}(M) \supseteq \mathsf{FV}(N)$ and a linear function f such that for every Γ, S and σ,

$$\langle \Gamma,\ M\sigma,\ S \rangle \Downarrow_{(h,s)} \implies \langle \Gamma,\ N\sigma,\ S \rangle \Downarrow_{(f(h),f(s))},$$

but where $M \overset{\not\rhd}{\approx} N$.

Proof. (Sketch) The result follows from the fact that

$$M \overset{\not\rhd}{\approx} \mathsf{let}\ \{y = x\}\ \mathsf{in}\ M[y/x].$$

Intuitively, this improvement cannot hold because if we apply the transformation in the body of a recursive function it could lead to repeated allocations of the binding for y which builds up a chain of closures. And the chain may grow with the depth of the recursion. But we can show that for every Γ and S

$$\begin{array}{l}\langle \Gamma,\ M,\ S \rangle \Downarrow_{(h,s)} \implies \\ \qquad \langle \Gamma,\ \mathsf{let}\ \{y = x\}\ \mathsf{in}\ M[y/x],\ S \rangle \Downarrow_{(h+1,s+1)}.\end{array}$$

The reason is simply that the modified term is executed exactly once so the binding may only be allocated once and thus can not lead to an arbitrary difference in space use. Thus the context lemma cannot hold since then we would have $M \overset{\rhd}{\approx} \mathsf{let}\ \{y = x\}\ \mathsf{in}\ M[y/x]$ which is not the case. □

Fixed Point Approximation It is typical in semantics to characterise recursion in terms of the "finite approximations" of recursive definitions. This approach is built in to the Scott-style denotational semantics approach where recursion is modelled by a least fixed point construction. The essence of this approach can be expressed in a purely operational setting. See e.g. [Smi92, MST96].

The natural formulation of the least fixed-point property also fails to hold for weak improvement. To make this claim precise we define the notion of the space-faithful n'th unwinding of a recursive definition. Let $\mathbb{V}$ range over value-contexts. We denote the n'th unwinding of a call to a function f in the context $\mathsf{let}\ \{f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[\cdot]$. by $\mathsf{let}\ \{f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^n]$. The details of the construction are given in Section 5. We are now ready to state our discontinuity result.

Theorem 4.5 (Syntactic Discontinuity of $\underset{\approx}{\rhd}$)
It is not always the case that $\mathsf{let}\ \{f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f]$ is the least upper bound of the chain

$$\mathsf{let}\ \{f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^0] \underset{\approx}{\rhd} \mathsf{let}\ \{f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^1] \underset{\approx}{\rhd} \ldots$$

Intuitively, syntactic continuity fails for the same reason as the context lemma. This is refelected in the similarity of the two proofs.

PROOF. (Sketch) It is possible to the sketch the proof of our claim without the exact definition of space-faithful finite unwindings. For this proof sketch we assume that there are unbounded integers in our language. The actual proof has to use an logarithmic encoding of unbounded integers through a list of bounded integers. The starting point of our proof sketch is the following two contextually equivalent definitions of a function f.

$$f = \lambda x.\lambda y.\mathsf{if}\ x \leq 0\ \mathsf{then}\ 1\ \mathsf{else}\ f\,(x-1)\,y$$

$$f = \lambda x.\lambda y.\mathsf{if}\ x \leq 0\ \mathsf{then}\ 1\ \mathsf{else}\ \mathsf{let}\ \{y' = y\}\ \mathsf{in}\ f\,(x-1)\,y'$$

The functions simply counts down their first argument until it is less or equal to zero and then returns one. The difference between them is that the second one builds a chain of bindings in the heap of length proportional to the depth of the recursion. As a result the second function can use arbitrary more heap depending on the value of the first argument so we can conclude that

$$\mathsf{let}\ \{f = \mathbb{V}_0[f]\}\ \mathsf{in}\ f \not\underset{\approx}{\rhd} \mathsf{let}\ \{f = \mathbb{V}_1[f]\}\ \mathsf{in}\ f$$

where $\mathbb{V}_0$ and $\mathbb{V}_1$ refers to the bodies of the first and the second definition respectively. [4] However even though the functions have asymptoticly different

[4] Note that this would not be true if we had not assumed unbounded integers since then the recursion depth would not exceed the largest integer.

space behaviours the finite unwindings of the two functions are related. More specificly, for every n,

$$\mathsf{let}\ \{f = \mathbb{V}_0[f]\}\ \mathsf{in}\ f^n \mathrel{\lessapprox\!\gtrapprox} \mathsf{let}\ \{f = \mathbb{V}_1[f]\}\ \mathsf{in}\ f^n.$$

The explanation is that a terminating computation which involves the n'th unwinding can not call the unwinding with a first argument greater than n. As a result the difference in space between the unwindings of the two definitions can not be greater than n so it is bounded by a constant. The fact that each unwinding of the two functions are related means that the chains of unwindings of the two definitions are identical up to $\lessapprox\!\gtrapprox$. So $\mathsf{let}\ \{f = \mathbb{V}_0[f]\}\ \mathsf{in}\ f$ and $\mathsf{let}\ \{f = \mathbb{V}_1[f]\}\ \mathsf{in}\ f$ can not both be least upper bounds of their respective chains since it would imply $\mathsf{let}\ \{f = \mathbb{V}_0[f]\}\ \mathsf{in}\ f \mathrel{\lessapprox\!\gtrapprox} \mathsf{let}\ \{f = \mathbb{V}_1[f]\}\ \mathsf{in}\ f$ which contradicts our earlier conclusion. □

5 Strong Improvement

The failure of the context lemma and the fixed-point approximation property gives a very concrete motivation for studying a stronger relation, *strong improvement*:

Definition 5.1 (Strong improvement)
M is *strongly improved by* N, written $M \mathrel{\gtrsim\!\!\!\rhd} N$, if for all $\mathbb{C}$, σ such that $\mathbb{C}[M\sigma]$ and $\mathbb{C}[N\sigma]$ are closed,

$$\mathbb{C}[M\sigma]\Downarrow_{(h,s)} \implies \mathbb{C}[N\sigma]\Downarrow_{(h,s)}.$$

We write $M \mathrel{\lessapprox\!\gtrapprox} N$ to mean that $M \mathrel{\gtrsim\!\!\!\rhd} N$ and $N \mathrel{\gtrsim\!\!\!\rhd} M$.

Although the definition of strong improvement is somewhat arbitrary – since it deals with constant factors for a high-level abstract machine – it provides a practical means to establish weak improvement laws, since whenever $M \mathrel{\gtrsim\!\!\!\rhd} N$ then clearly $M \mathrel{\gtrapprox\!\!\!\rhd} N$. In this section we present some basic laws of strong improvement, and our key technical results: a context lemma and a fixed-point approximation theorem for establishing improvement properties of recursive definitions. The fixed-point approximation theorem for strong improvement coupled with the fact that fixed point induction fails for weak improvement puts strong improvement in focus – most of our calculations will concern strong improvement.

5.1 A Context Lemma

For strong improvement we have established a *context lemma* [Mil77]: to prove that M is strongly improved by N, one only needs to compare their behaviour with respect to a much smaller set of contexts, namely the context which immediately need to evaluate their holes.

Lemma 5.2 (Context Lemma)
For all M and N such that $\mathsf{FV}(M) \supseteq \mathsf{FV}(N)$, if for all Γ, S and σ,

$$\langle \Gamma,\ M\sigma,\ S\rangle \Downarrow_{h,s} \implies \langle \Gamma,\ N\sigma,\ S\rangle \Downarrow_{h,s}$$

then $M \underset{\sim}{\trianglerighteq} N$.

The proof requires a degree of technical machinery so we postpone it to Section 7. The context lemma gives us a way to prove basic laws of weak and strong improvement. To prove, for example, (restricted) beta-reduction $(\lambda x.M)\, y \underset{\approx}{\trianglerighteq} M[y/x]$ we will show the stronger property: $(\lambda x.M)\, y \underset{\sim}{\trianglerighteq} M[y/x]$. The context lemma makes this property very easy to establish (see Section 8.1 for a detailed proof of a generalised statement). The converse direction also holds within a constant factor under the assumption that y occurs free in $M[y/x]$. The only difference when going from the right-hand side to the left is that the left hand side will momentarily use up one stack unit more than the right-hand side.

In order to express the latter property using the more precise improvement theory, we need some space analogue of the time-tick from [MS99a]. In fact, we will use several kinds of "tick", which we we call the *space gadgets*.

5.2 The Space Gadgets

The *space gadgets* are syntactic means to represent and control the space properties of terms. They play a crucial role in strong improvement calculations. We describe each gadget in turn.

Dummy References The use of dummy references allows one to make assertions about, and to modify the liveness properties of variables. To this end we introduce the following notational extension, terms of the form ${}^{X}M$ where X is a multiset of variables. The construct is representable in the language and is defined thus

$${}^{\{\vec{x}\}}M \stackrel{\text{def}}{=} \mathsf{let}\ \{\vec{y} = \vec{x}\}\ \mathsf{in}\ M \qquad \text{where } \vec{y} \text{ are fresh.}$$

Hence ${}^{X}M$ behaves as M but in addition holds on to the variables in X until the evaluation of M starts.

Dummy references can express certain liveness properties. For example, if $\mathbb{C}[M] \underset{\sim}{\trianglerighteq} \mathbb{C}[{}^{\{y\}}M]$ then we know that y is still live at the occurrence of M. Among other things we will use dummy references to control the life time of *dummy bindings*, i.e., bindings which play no rôle in the term but to take up space. To add dummy bindings is harmless in the weak theory as long as their life time is coupled to another binding.

Lemma 5.3 (Dummy Binding Introduction)

$$\mathsf{let}\ \{x = M\}\ \mathsf{in}\ N \underset{\approx}{\trianglelefteq} \mathsf{let}\ \{z = \Omega, x = {}^{\{z\}}M\}\ \mathsf{in}\ N, \qquad \text{if } z \text{ is fresh}$$

PROOF. We only sketch the right-way improvement. The left-way improvement follows in a similar manner. For arbitrary $\mathbb{C}$ and σ such that $\mathbb{C}[(\mathsf{let}\ \{x = M\}\ \mathsf{in}\ N)\sigma]$ is closed, assume

$$\mathbb{C}[(\mathsf{let}\ \{x = M\}\ \mathsf{in}\ N)\sigma]\Downarrow^n_{(h,s)}.$$

By induction over the length of the computation we construct the corresponding transition sequence for $\mathbb{C}[(\mathsf{let}\ \{z = \Omega, x = {}^{\{z\}}M\}\ \mathsf{in}\ N)\sigma]\Downarrow^n_{(h,s)}$ maintaining the invariant that for each instance of the binding for z there is a corresponding instance of the binding for x. Thus the computation can take up at most twice the amount of heap space. □

Spikes Spikes are amortisation device which allow us to represent a very short-lived space usage – a spike in the space-usage profile. Spikes come in two varieties, *heap* spikes and *stack* spikes. The stack spike is defined thus

$$^{\curlyvee}M \stackrel{\text{def}}{=} \mathsf{case\ true\ of}\ \{\mathsf{true} \to M\}$$

It has the short-lived effect of increasing the stack usage by one unit, at the moment that M is about to be evaluated. The *heap spike* is the heap analogue of the stack spike; it momentarily increases the size of the heap at the point in time when the term is ready to be evaluated.

$$^{\curlywedge}M \stackrel{\text{def}}{=} \mathsf{let}\ x = \Omega\ \mathsf{in}\ {}^{\{x\}}M \qquad \text{where } x \text{ is fresh}$$

To see how spikes are used, consider how we prove (restricted) beta-expansion,

$$M[y/x] \gtrapprox (\lambda x.M)\, y \qquad \text{if } y \in \mathsf{FV}(M[y/x]).$$

The difference between the terms is that the right hand side will momentarily use up one stack unit more than the right-hand side. We can compensate for it with a stack spike and prove that

$$^{\curlyvee}M[y/x] \gtrsim (\lambda x.M)\, y \qquad \text{if } y \in \mathsf{FV}(M[y/x]).$$

which is easy using the context lemma (see Section 8.1 for a detailed proof of a generalised statement). All that is left is to establish that spike introduction is harmless in the weak theory:

Lemma 5.4 (Spike Introduction)

1. $M \lessapprox {}^{\curlyvee}M$
2. $M \lessapprox {}^{\curlywedge}M$

The proof is analogous to that for time ticks in [MS99a].

Weights The most complex gadgets are the weights[5]. Weights are more involved because they cannot be defined in terms of existing language constructs, but must be added as a collection of term-annotations with a specially defined space-semantics.

In our definition of space use we count every entity on the stack or on the heap as occupying exactly one unit of space, a choice justified by our desire to ultimately reason about asymptotic behaviour. But it turns out to be crucial to be able to selectively choose exactly how much space each entity shall account for – i.e., what the *weight* of the entity should be. Consider, for example, the following weak equivalence law for reduction contexts (defined in Section 2.1 and ranged over by R):

$$R[\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}] \mathrel{\lhd\!\!\rhd}_{\approx} \mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to R[N_i]\}$$

It is not a strong space equivalence since the left hand side takes up more space: while M is being evaluated, both R and the case-alternatives take up stack space (2 units of space). In the right hand side, while M is being evaluated there is just a single set of case alternatives (1 unit of stack space). We can compensate for this, and simplify our calculations, if we count the case in the right hand side as occupying two units of stack, which we denote by the following weight annotation:

$$R[\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}] \mathrel{\lhd\!\!\rhd}_{\sim} {}^{2}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to R[N_i]\}$$

This is not the only form of weight, but before we consider further examples we will give the semantics of weights.

We will annotate every entity on the heap and the stack with a weight $w \geq 0$. Binding occurrences of variables, including update markers (which are considered to take up both heap and stack space) are annotated with two weights, one for the heap and one for the stack. The space consumption of each entity is given by the following:

$$|{}^{w}_{v}x = M| = (v, 0) \qquad |{}^{w}R| = (0, w) \qquad |\#^{w}_{v}x| = (v, w)$$

So the upper weight of the binder is the stack weight, incurred when the update marker is on the stack; the lower weight is the heap weight – the size of the binding on the heap.

Note that weights may be zero so we can specify that an entity shouldn't be counted for at all. An entity without a weight annotation will now be taken as shorthand for a weight of 1. The weight on bindings and stack elements originate from annotation in the program. Our annotated term language is

$$\begin{array}{rcl} L, M, N & ::= & x \mid \lambda x.M \mid {}^{w}(M\ x) \mid c\,\vec{x} \mid {}^{w}(\mathsf{seq}\ M\ N) \\ & \mid & n \mid {}^{w_0}(M +^{w_1} N) \mid {}^{w}(\mathsf{add}_n M) \mid {}^{w}(\mathsf{iszero}\ M) \\ & \mid & \mathsf{let}\ \{{}^{v_i}_{w_i}x_i = M_i\}_{i\in I}\ \mathsf{in}\ N \\ & \mid & {}^{w}(\mathsf{case}\ M\ \mathsf{of}\ \{c_i\,\vec{x}_i \to N_i\}). \end{array}$$

[5] A generalisation of the ballasts from [GS99].

$$\langle\, \Gamma\{{}^{v}_{w} x = M\},\ x,\ S \,\rangle \to \langle\, \Gamma,\ M,\ \#^{v}_{w} x : S \,\rangle \qquad (\mathit{Lookup})$$

$$\langle\, \Gamma,\ V,\ \#^{v}_{w} x : S \,\rangle \to \langle\, \Gamma\{{}^{v}_{w} x = V\},\ V,\ S \,\rangle \qquad (\mathit{Update})$$

$$\langle\, \Gamma,\ \mathsf{let}\ \Gamma'\ \mathsf{in}\ N,\ S \,\rangle \to \langle\, \Gamma\Gamma',\ N,\ S \,\rangle \qquad (\mathit{Letrec})$$

$$\langle\, \Gamma,\ {}^{w}R[M],\ S \,\rangle \to \langle\, \Gamma,\ M,\ {}^{w}R : S \,\rangle \qquad (\mathit{Push})$$

$$\langle\, \Gamma,\ V,\ {}^{w}R : S \,\rangle \to \langle\, \Gamma,\ M,\ S \,\rangle \quad \text{if } R[V] \rightsquigarrow M \qquad (\mathit{Reduce})$$

$$(\lambda x.M)\, y \rightsquigarrow M\,[{}^{y}/_{x}]$$

$$\mathsf{case}\ c_j\, \vec{y}\ \mathsf{of}\ \{c_i\, \vec{x}_i \to M_i\} \rightsquigarrow M_j[{}^{\vec{y}}/_{\vec{x}_j}]$$

$$\mathsf{seq}\, V\, M \rightsquigarrow M$$

$$m +^{w} N \rightsquigarrow {}^{w}(\mathsf{add}_m N)$$

$$\mathsf{add}_m\, n \rightsquigarrow \ulcorner m + n \urcorner$$

$$\mathsf{iszero}\, m \rightsquigarrow \begin{cases} \mathsf{true} & \text{if } m = 0 \\ \mathsf{false} & \text{otherwise} \end{cases}$$

Figure 2: Abstract machine semantics with weights

In Figure 2 we have extended the abstract machine rules with weights.

Of course, weights have no intrinsic interest for programmers – they are a bookkeeping mechanism which we use to syntactically account for certain forms of space usage. As with spikes, a crucial property of weights is that they increase space use in the strong theory but do not change space behaviour by more than a constant factor:

Lemma 5.5
If $w \geq w'$ and $v \geq v'$ then

1. ${}^{w}R[M] \trianglerighteq {}^{w'}R[M]$.
2. ${}^{w}(M +^{v} N) \trianglerighteq {}^{w'}(M +^{v'} N)$.
3. $\mathsf{let}\ \Gamma\{{}^{w}_{v} x = M\}\ \mathsf{in}\ N \trianglerighteq \mathsf{let}\ \Gamma\{{}^{w'}_{v'} x = M\}\ \mathsf{in}\ N$.

Proof. We only sketch the proof of 1 the others follow similarly. Assume $w \geq w'$. For arbitrary $\mathbb{C}$ and σ such that $\mathbb{C}[{}^{w}R[M]\sigma]$ is closed, assume

$$\mathbb{C}[{}^{w}R[M]\sigma]\Downarrow^{n}_{(h,s)}.$$

By induction over the length of the computation we construct the corresponding transition sequence for $\mathbb{C}[{}^{w'}R[M]\sigma]\Downarrow^{n}_{(h,s)}$ maintaining the invariant that the

weights in configurations are less or equal to the weights in the original transition sequence. □

Lemma 5.6
For $v, w > 0$,

1. $R[M] \mathrel{\lessapprox\!\gtrapprox} {}^{w}R[M]$.
2. $(M + N) \mathrel{\lessapprox\!\gtrapprox} {}^{w}(M +^{v} N)$.
3. $\mathsf{let}\ \Gamma\{x = M\}\ \mathsf{in}\ N \mathrel{\lessapprox\!\gtrapprox} \mathsf{let}\ \Gamma\{{}^{w}_{v}x = M\}\ \mathsf{in}\ N$.

Proof. The proof is very similar to the proof of the previous lemma but maintains the invariant that the weights in the constructed transition sequence is within the constant factor. □

Zero weights or "balloons" play a special role, and must be handled with care. A zero weight permits costs to be hidden. This is very useful in strong improvement calculations since it cuts down significantly on the "noise" of weight bookkeeping. However, adding zero-weights is potentially unsound, since we might end up hiding an asymptotic amount of space usage. In other words, we cannot arbitrarily introduce zero weights in the weak improvement theory (c.f. Lemma 5.6). There are two ways in which we can justify zero-weight introduction. The first is if an entity is short-lived so that it can't affect the asymptotic space behaviour. We will heavily use two instances of this: that the weight of a stack frame associated with an application of a known function can be ignored and that the update marker weight of a value binding can be safely ignored. This is because its lifetime on the stack is only one computation step.

Lemma 5.7 (Balloon introduction)

1. $(\lambda x.M)\, y \mathrel{\lessapprox\!\gtrapprox} {}^{0}((\lambda x.M)\, y)$
2. $\mathsf{let}\ \{x = V\}\ \mathsf{in}\ N \mathrel{\lessapprox\!\gtrapprox} \mathsf{let}\ \{{}^{0}x = V\}\ \mathsf{in}\ N$

We will use zero-weights on applications often so we introduce an abbreviation and write $M \cdot x$ for ${}^{0}(M\ x)$. The zero weights on applications cuts down significantly on the "noise" from spikes in strong improvement calculations because of the law

$$(\lambda x.M) \cdot y \mathrel{\lessapprox} M[{}^{y}/_{x}] \qquad \text{if } y \in \mathsf{FV}(M[{}^{y}/_{x}]).$$

The law is a strong space equivalence and we do not need a spike to compensate for the short-lived entity on the stack because it doesn't account for any space. With this law we can prove part 1 of Lemma 5.7 under the additional assumption that $y \in \mathsf{FV}(M[{}^{y}/_{x}])$. With the laws presented in Section 5.3 it easy to lift the restriction. The proof is by a simple calculation:

$$(\lambda x.M)\, y \mathrel{\lessapprox} {}^{\gamma}M[{}^{y}/_{x}] \mathrel{\lessapprox\!\gtrapprox} M[{}^{y}/_{x}] \mathrel{\lessapprox} (\lambda x.M) \cdot y$$

The second way that we introduce zero weights is via a "top-level" assumption. It is safe to introduce zero weights to bindings which will not be allocated multiply. Unfortunately this is not a property that holds in all contexts, but is still reasonable. For example, functions from a standard library are typically allocated just once – i.e. they are top level definitions. If a function is defined at top-level then setting heap-weight to zero can have at most a constant factor effect:

Lemma 5.8
For every Γ there exist k such that for every M, if $\mathsf{let}\ \{\Delta\}\ \mathsf{in}\ M\Downarrow_{(h,s)}$ then $\mathsf{let}\ \{\Gamma\}\ \mathsf{in}\ M\Downarrow_{(h+k,s)}$, where Δ is the result of setting all heap weights on bindings in Γ to zero.

PROOF. By induction over the length of the computation □

To see why we need zero heap weights on bindings consider the problem of showing an improvement of the form

for every $\mathbb{C}$,

$$\mathsf{let}\ \Gamma\{f = V, g = W\}\ \mathsf{in}\ \mathbb{C}[f\,x] \lessapprox \mathsf{let}\ \Gamma\{f = V, g = W\}\ \mathsf{in}\ \mathbb{C}[g\,x].$$

The statement is typically not true even if f and g have the same space behaviour. The argument is similar to the proof of the free variable restriction theorem for weak improvement: $f\,x$ holds on to a reference to f but $g\,x$ holds on to a reference to g. Suppose we construct a context $\mathbb{C}$ such that $\mathbb{C}$ never evaluates the hole and holds on to f but not g. Then $\mathbb{C}[f\,x]$ holds on to only f but $\mathbb{C}[g\,x]$ holds on to f and g and can thus prevent the garbage collection of g. It does not necessarily lead to a different space behaviour though: if there, for example, is a reference to g in the body of f or in a function f refers to then g cannot be collected in any case. But for many definitions of f and g the argument is valid and the desired improvement does not hold. Our pragmatic solution is to instead prove an improvement of the form

for every $\mathbb{C}$,

$$\mathsf{let}\ \Gamma\{{}_0f = V, {}_0g = W\}\ \mathsf{in}\ \mathbb{C}[f\,x] \lessapprox \mathsf{let}\ \Gamma\{{}_0f = V, {}_0g = W\}\ \mathsf{in}\ \mathbb{C}[g\,x]$$

where we have put a zero heap weight on the bindings for f and g. The improvement can be applied in any context but ultimately the zero weight has to be justified in some way. For example by f and g being top level definitions in the program in question.

Finally, we note that with the help of weights we can increase the size of the stack and heap spikes:

$$^{w\,\curlyvee}M \stackrel{\text{def}}{=} {}^{w}\mathsf{case\ true\ of}\ \{\mathsf{true} \to M\}$$
$$^{w\,\curlywedge}M \stackrel{\text{def}}{=} \mathsf{let}\ {}_{w}x = \Omega\ \mathsf{in}\ {}^{\{x\}}M \qquad \text{where } x \text{ is fresh}$$

5.3 Laws of strong improvement

Now that we have our space gadgets we will use them to state a collection of laws for strong improvement. Like any other contextual program ordering, it is not recursively enumerable, so any such collection is inevitably somewhat ad hoc. In presenting the laws, we will follow the standard free-variable convention [Bar81] that all bound variables in the statement of a law are distinct, and that they are disjoint from the free variables. Later, in Section 8, when we have introduced some technical machinery we will prove some of the laws in detail.

Reduction The most fundamental law is *reduction*:

$$ {}^{w}R[V] \mathrel{\lessapprox\!\!\gtrsim} {}^{w\gamma X}N \qquad \text{if } R[V] \rightsquigarrow N \text{ and } \mathsf{FV}(R[V]) = \mathsf{FV}({}^{X}N) \qquad (\textit{reduction}) $$

which subsumes the beta-reduction law presented earlier. Recall that the spike ${}^{w\gamma}$ in the right hand side is there to make up for the additional stack used in the left hand side. A key property of the law is that it can be applied even if we discard variables when we perform the redex because we may compensate through dummy references in X. In practice, redexes which discards variables often shows up in reductions of case expressions. For example if we reduce

$$ \mathsf{case}\ x : xs\ \mathsf{of}\ \{\mathsf{nil} \to M, y : ys \to N\} $$

the free variables of M is lost so we must put them in X in order to apply the law. With *reduction* we can prove part 1 of Lemma 5.7, i.e., that

$$ (\lambda x.M)\, y \mathrel{\lessapprox\!\!\gtrsim} (\lambda x.M) \cdot y $$

without the additional side condition that $y \in \mathsf{FV}(M[{}^{y}/_{x}])$. The derivation goes as follows.

$$ (\lambda x.M)\, y \mathrel{\lessapprox\!\!\gtrsim} {}^{\gamma\{y\}}M[{}^{y}/_{x}] \mathrel{\lessapprox\!\!\gtrsim} {}^{0\gamma\{y\}}M[{}^{y}/_{x}] \mathrel{\lessapprox\!\!\gtrsim} (\lambda x.M) \cdot y $$

Unfolding For unfolding of values we have the following weak space equivalence.

$$ \mathsf{let}\ \Gamma\{{}^{v}_{w}x = V\}\ \mathsf{in}\ \mathbb{C}[x] \mathrel{\lessapprox\!\!\gtrsim} \mathsf{let}\ \Gamma\{{}^{v}_{w}x = V\}\ \mathsf{in}\ \mathbb{C}[{}^{\{x\}}V] $$

The dummy reference to x in the right hand side is there to ensure that the binding for x is not garbage collected earlier because of the unfolding. However the transformation is not a strong improvement because the left hand side may momentarily use v extra units of stack. The stack use comes from the update marker with weight v that is pushed on the stack when x is looked up. Because x is bound to a value the update marker is short-lived so it may seem as if it could be modelled by a spike in the right hand side of the law. However, the update marker may be garbage collected if there are no other occurrences of

x. If so the update marker need not account for any space. For this reason we cannot compensate for the update marker by adding a spike to the right hand side. Instead we can show the weak improvement by first showing the two strong improvements

$$\mathsf{let}\ \Gamma\{{}^{v}_{w}x = V\}\ \mathsf{in}\ \mathbb{C}[x] \trianglerighteq \mathsf{let}\ \Gamma\{{}^{v}_{w}x = V\}\ \mathsf{in}\ \mathbb{C}[{}^{\{x\}}V], \qquad (\textit{unfold})$$

and

$$\mathsf{let}\ \Gamma\{{}^{v}_{w}x = V\}\ \mathsf{in}\ \mathbb{C}[x] \trianglelefteq \mathsf{let}\ \Gamma\{{}^{v}_{w}x = V\}\ \mathsf{in}\ \mathbb{C}[{}^{\{x\}v\curlyvee}V]. \qquad (\textit{fold})$$

and then use the spike introduction lemma (Lemma 5.4).

In calculations it is a nuisance to have two rules – to show an equivalence we may need to make two derivations, one in each direction. However, in some special cases we have a strong equivalence. One such case is when the update marker weight is zero so that it does not take up any space. In this case the two improvements turns into an equivalence.

$$\mathsf{let}\ \Gamma\{{}^{0}_{w}x = V\}\ \mathsf{in}\ \mathbb{C}[x] \trianglerighteq\!\trianglelefteq \mathsf{let}\ \Gamma\{{}^{0}_{w}x = V\}\ \mathsf{in}\ \mathbb{C}[{}^{\{x\}}V]$$

Another case is when the definition of x is recursive. Then the update marker cannot be garbage collected so we can compensate with a stack spike in the right hand side:

$$\mathsf{let}\ \Gamma\{{}^{v}_{w}x = V\}\ \mathsf{in}\ \mathbb{C}[x] \trianglerighteq\!\trianglelefteq \mathsf{let}\ \Gamma\{{}^{v}_{w}x = V\}\ \mathsf{in}\ \mathbb{C}[{}^{v\curlyvee}V] \qquad \text{if } x \in \mathsf{FV}(V)$$

Note also that there is no need to keep a dummy reference to x in the right hand side because $x \in \mathsf{FV}(V)$. Our last rule for unfolding is for bindings with a zero stack weight and a zero heap weight.

$$\mathsf{let}\ \Gamma\{{}^{0}_{0}x = V\}\ \mathsf{in}\ \mathbb{C}[x] \trianglerighteq\!\trianglelefteq \mathsf{let}\ \Gamma\{{}^{0}_{0}x = V\}\ \mathsf{in}\ \mathbb{C}[V]$$

In this rule there is no need for a dummy reference to x in the right hand side, because the binding takes up no space. This is the rule that we will use mostly in actual strong improvement calculations because most unfolding steps concerns unfolding a top-level definition. We have collected the unfolding rules in Figure 3.

Rules for let We have collected the rules for manipulating let-expressions in Figure 4. The first law

$$\mathsf{let}\ \{{}^{v}_{w}x = M\}\ \mathsf{in}\ x \trianglerighteq\!\trianglelefteq {}^{w\curlywedge}M, \qquad (\textit{let-elim})$$

eliminates a let-expression. We use it intensively in calculations where a function takes a non-value as an argument – recall that when N is not a variable an application $M\,N$ is syntactic sugar for $\mathsf{let}\ \{x = N\}\ \mathsf{in}\ M\,x$ so all arguments to functions are bound in a let-expression. If the argument is a value we can use

$$\mathsf{let}\ \Gamma\{^v_w x = V\}\ \mathsf{in}\ \mathbb{C}[x] \gtrsim \mathsf{let}\ \Gamma\{^v_w x = V\}\ \mathsf{in}\ \mathbb{C}[^{\{x\}}V]$$

$$\mathsf{let}\ \Gamma\{^v_w x = V\}\ \mathsf{in}\ \mathbb{C}[x] \lesssim \mathsf{let}\ \Gamma\{^v_w x = V\}\ \mathsf{in}\ \mathbb{C}[^{\{x\}v}{}^{\gamma}V]$$

$$\mathsf{let}\ \Gamma\{^v_w x = V\}\ \mathsf{in}\ \mathbb{C}[x] \lessapprox \mathsf{let}\ \Gamma\{^v_w x = V\}\ \mathsf{in}\ \mathbb{C}[^{v}{}^{\gamma}V] \qquad \text{if } x \in \mathsf{FV}(V)$$

$$\mathsf{let}\ \Gamma\{^0_w x = V\}\ \mathsf{in}\ \mathbb{C}[x] \lessapprox \mathsf{let}\ \Gamma\{^0_w x = V\}\ \mathsf{in}\ \mathbb{C}[^{\{x\}}V]$$

$$\mathsf{let}\ \Gamma\{^0_0 x = V\}\ \mathsf{in}\ \mathbb{C}[x] \lessapprox \mathsf{let}\ \Gamma\{^0_0 x = V\}\ \mathsf{in}\ \mathbb{C}[V]$$

Figure 3: Unfolding laws.

one of the rule for unfolding a value but if it is not a value we are left with *let-elim*. To get the opportunity to apply the law we need to float the let to the place of use. The laws *let-R*, *let-flatten*, *let-let*, *let-alts*, *let-let'*, *let-alts'* are for this purpose. The laws are quite restricted because when we move a let it may lead to later or earlier allocation. For example in *let-let*, *let-alts*, *let-let'*, *let-alts'* we can delay the allocation of the bindings in Γ only if we compensate and allocate the bindings in Δ earlier. To be able to apply the rules we often need to introduce allocations of dummy bindings which we float around to compensate for the floating of the other bindings (see Section 6 for some examples). Another restriction on floating lets is that if a function uses an argument twice it will typically not be possible to float the corresponding let to both places since it could lead to a duplication of computation – which often is not space safe. The exception is if the two uses are in two different branch of a case expression. Then we can often use *let-alts* or *let-alts'*. Also, if the binding binds a value we may copy the binding and float one copy to each place. The rules *value-merge*, *value-copy*, *value-merge'*, *value-copy'* allows value bindings to be copied or merged. Finally, the rule

$$\mathsf{let}\ \Gamma\Gamma'\ \mathsf{in}\ M \lessapprox {}^{X}\mathsf{let}\ \{\Gamma\}\ \mathsf{in}\ M \qquad \text{if } \mathsf{FV}(\mathsf{let}\ \Gamma\Gamma'\ \mathsf{in}\ M) = \mathsf{FV}({}^{X}\mathsf{let}\ \{\Gamma\}\ \mathsf{in}\ M) \quad (gc)$$

allows bindings in lets to be removed analogous to garbage collection. Note that it is implicit by the free variable convention that we cannot remove bindings which there is a reference to because it would cause a bound variable to become free. If we remove a binding that has free variables then we must take care to compensate by putting the free variables in X so that both sides have the same free variables. Otherwise, by the free variable restriction theorem the two terms cannot be space equivalent. The rule *empty-let* let us remove an empty let which can be the result of applying the garbage collection rule.

$$\mathsf{let}\ \{^{v}_{w}x = M\}\ \mathsf{in}\ x \mathrel{\mathop{\lesssim}^{\Phi}} {}^{w\searrow}M \qquad (\textit{let-elim})$$

$$\mathsf{let}\ \Gamma\ \mathsf{in}\ {}^{w}R[M] \mathrel{\mathop{\lesssim}^{\Phi}} {}^{w}R[\mathsf{let}\ \Gamma\ \mathsf{in}\ M] \qquad \text{if } \operatorname{dom}\Gamma \subseteq \mathsf{FV}(M) \quad (\textit{let-R})$$

$$\mathsf{let}\ \Gamma\ \mathsf{in}\ \mathsf{let}\ \Delta\ \mathsf{in}\ M \mathrel{\mathop{\lesssim}^{\Phi}} \mathsf{let}\ \Gamma\Delta\ \mathsf{in}\ M \quad \text{if } \operatorname{dom}\Delta \subseteq \mathsf{FV}(M) \quad (\textit{let-flatten})$$

$$\begin{aligned}&\mathsf{let}\ \Gamma\{^{v}_{w}x = \mathsf{let}\ \Delta\ \mathsf{in}\ M\}\ \mathsf{in}\ N \qquad (\textit{let-let})\\ &\quad\mathrel{\mathop{\lesssim}^{\Phi}} \mathsf{let}\ \Delta\{^{v}_{w}x = \mathsf{let}\ \Gamma\ \mathsf{in}\ M\}\ \mathsf{in}\ N\\ &\qquad\text{if } \operatorname{dom}\Gamma \cup \operatorname{dom}\Delta \subseteq \mathsf{FV}(M), \text{ and } |\Gamma| = |\Delta|\end{aligned}$$

$$\begin{aligned}&\mathsf{let}\ \Gamma\ \mathsf{in}\ {}^{w}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Delta\ \mathsf{in}\ N_i\} \qquad (\textit{let-alts})\\ &\quad\mathrel{\mathop{\lesssim}^{\Phi}} \mathsf{let}\ \Delta\ \mathsf{in}\ {}^{w}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Gamma\ \mathsf{in}\ N_i\}\\ &\qquad\text{if } \operatorname{dom}\Gamma \cup \operatorname{dom}\Delta \subseteq \mathsf{FV}(N_i), \text{ and } |\Gamma| = |\Delta|\end{aligned}$$

$$\begin{aligned}&\mathsf{let}\ \Gamma\{^{v}_{w}x = M\}\ \mathsf{in}\ N \mathrel{\mathop{\lesssim}^{\Phi}} \mathsf{let}\ \Delta\{^{v}_{w}x = {}^{\operatorname{dom}\Delta}\mathsf{let}\ \Gamma\ \mathsf{in}\ M\}\ \mathsf{in}\ N \qquad (\textit{let-let'})\\ &\qquad\text{if } \operatorname{dom}\Gamma \subseteq \mathsf{FV}(M), \text{ and } |\Gamma| = |\Delta|\end{aligned}$$

$$\begin{aligned}&\mathsf{let}\ \Gamma\ \mathsf{in}\ {}^{w}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\} \qquad (\textit{let-alts'})\\ &\quad\mathrel{\mathop{\lesssim}^{\Phi}} \mathsf{let}\ \Delta\ \mathsf{in}\ {}^{w}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to {}^{\operatorname{dom}\Delta}\mathsf{let}\ \Gamma\ \mathsf{in}\ N_i\}\\ &\qquad\text{if } \operatorname{dom}\Gamma \subseteq \mathsf{FV}(N_i),\ |\Gamma| = |\Delta|\end{aligned}$$

$$\mathsf{let}\ \Gamma\{x = V,\ y = V\}\ \mathsf{in}\ M \mathrel{\mathop{\gtrsim}^{\Phi}} \mathsf{let}\ \Gamma[{}^{x}/_{y}]\{x = V[{}^{x}/_{y}]\}\ \mathsf{in}\ M[{}^{x}/_{y}] \qquad (\textit{value-merge})$$

$$\mathsf{let}\ \Gamma\{x = V,\ y = V\}\ \mathsf{in}\ M \mathrel{\mathop{\lesssim}^{\Phi}} \mathsf{let}\ \Gamma[{}^{x}/_{y}]\{{}_{2}x = V[{}^{x}/_{y}]\}\ \mathsf{in}\ M[{}^{x}/_{y}] \qquad (\textit{value-copy})$$

$$\begin{aligned}&\mathsf{let}\ \Gamma\{x = \mathsf{let}\ \{y = V\}\ \mathsf{in}\ V\}\ \mathsf{in}\ M\\ &\quad\mathrel{\mathop{\gtrsim}^{\Phi}} \mathsf{let}\ \Gamma\{x = V[{}^{x}/_{y}]\}\ \mathsf{in}\ M \qquad (\textit{value-merge'})\end{aligned}$$

$$\begin{aligned}&\mathsf{let}\ \Gamma\{x = \mathsf{let}\ \{y = V\}\ \mathsf{in}\ V\}\ \mathsf{in}\ M\\ &\quad\mathrel{\mathop{\lesssim}^{\Phi}} \mathsf{let}\ \Gamma\{{}_{2}x = V[{}^{x}/_{y}]\}\ \mathsf{in}\ M \qquad (\textit{value-copy'})\end{aligned}$$

$$\begin{aligned}&\mathsf{let}\ \Gamma\Gamma'\ \mathsf{in}\ M \mathrel{\mathop{\lesssim}^{\Phi}} {}^{X}\mathsf{let}\ \Gamma\ \mathsf{in}\ M \qquad (\textit{gc})\\ &\qquad\text{if } \mathsf{FV}(\mathsf{let}\ \Gamma\Gamma'\ \mathsf{in}\ M) = \mathsf{FV}({}^{X}\mathsf{let}\ \{\Gamma\}\ \mathsf{in}\ M)\end{aligned}$$

$$\mathsf{let}\ \{\}\ \mathsf{in}\ N \mathrel{\mathop{\lesssim}^{\Phi}} N \qquad (\textit{empty-let})$$

Figure 4: Laws for lets.

$$ {}^{w}R[{}^{v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}] \qquad (R\text{-}case) $$
$$ \lhd\!\rhd_{\sim} {}^{w+v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to {}^{w}R[N_i]\} $$

$$ \mathsf{case}\ x\ \mathsf{of}\ \{alts, c\,\vec{y} \to \mathbb{D}[x]\} \qquad (case\text{-}unfold) $$
$$ \rhd_{\sim} \mathsf{case}\ x\ \mathsf{of}\ \{alts, c\,\vec{y} \to \mathbb{D}[{}^{\{x\}}c\,\vec{y}]\} $$

$$ \mathsf{let}\ \{{}^{v}_{w}x = M\}\ \mathsf{in}\ \mathbb{C}[\mathsf{case}\ x\ \mathsf{of}\ \{alts, c\,\vec{y} \to \mathbb{D}[x]\}] \qquad (case\text{-}fold) $$
$$ \lhd_{\sim} \mathsf{let}\ \{{}^{v}_{w}x = M\}\ \mathsf{in}\ \mathbb{C}[\mathsf{case}\ x\ \mathsf{of}\ \{alts, c\,\vec{y} \to \mathbb{D}[{}^{\{x\}_{v\,\curlyvee}}c\,\vec{y}]\}] $$

Figure 5: Laws for case.

Rules for case In Figure 5 we have collected some rules for case-expressions. The first law

$$ {}^{w}R[{}^{v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}] \lhd\!\rhd_{\sim} {}^{w+v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to {}^{w}R[N_i]\} \qquad (R\text{-}case) $$

is a generalisation of the classical case-of-case rule. It allows an arbitrary reduction context to be floated into the branches of a case. Note that we have to add extra weight on the case in the right hand side to make the rule a strong improvement. Consider the next law:

$$ \mathsf{case}\ x\ \mathsf{of}\ \{alts, c\,\vec{y} \to \mathbb{D}[x]\} \rhd_{\sim} \mathsf{case}\ x\ \mathsf{of}\ \{alts, c\,\vec{y} \to \mathbb{D}[{}^{\{x\}}c\,\vec{y}]\} \qquad (case\text{-}unfold) $$

If a variable x is scrutinised by a case expression and x occurs in one of the branches we may replace the x in the branch with the value corresponding to the pattern in the branch provided we keep a dummy reference to x. Note that it is not possible to extend the rule to case expressions which scrutinise an arbitrary term N because it would potentially introduce additional sharing of computation which can lead to an asymptotically different space behaviour. The next rule *case-fold* is an adaption of *case-unfold* where the improvement goes in the other direction. These two rules suffer from the same problem due to the update marker as *unfold* and *fold*. Just like for *unfold* and *fold*, if the update marker weight on the binding for x is zero the two rules coincide into an equivalence.

Rules for Ω In Figure 6 we have collected some laws for the divergent term $\mathsf{let}\ \{x = x\}\ \mathsf{in}\ x$ which we denote by Ω. These laws are used extensively when establishing the base case in a proof based on fixed point induction.

The Spike Algebra Spikes are a nuisance in calculations with strong improvement – they often get in the way of applying other rules. The Spike Algebra in Figure 7 lets us eliminate and move away spikes. In calculations we will sometimes silently apply the Spike Algebra.

$$
\begin{aligned}
{}^{X}\Omega &\gtrsim M \qquad \text{if } \mathsf{FV}(M) \subseteq X \\
\mathsf{let}\ \Gamma\{x = {}^{X}\Omega\}\ \mathsf{in}\ N &\gtrsim \mathsf{let}\ \Gamma\{x = M\}\ \mathsf{in}\ N \qquad \text{if } \mathsf{FV}(M) \subseteq X \cup \{x\} \\
R[\Omega] &\lesssim\gtrsim {}^{\mathsf{FV}(R)}\Omega \\
\mathsf{let}\ \Gamma\ \mathsf{in}\ {}^{X}\Omega &\lesssim\gtrsim {}^{Y}\Omega \qquad \text{if } Y = \mathsf{FV}(\mathsf{let}\ \Gamma\ \mathsf{in}\ {}^{X}\Omega) \\
{}^{w\curlyvee}\Omega &\lesssim\gtrsim \Omega \\
{}^{w\curlywedge}\Omega &\lesssim\gtrsim \Omega \\
\mathsf{let}\ \Gamma\{{}^{v}_{w}x = \Omega\}\ \mathsf{in}\ \mathbb{C}[x] &\lesssim\gtrsim \mathsf{let}\ \Gamma\{{}^{v}_{w}x = \Omega\}\ \mathsf{in}\ \mathbb{C}[{}^{\{x\}}\Omega]
\end{aligned}
$$

Figure 6: Rules for Ω.

The Dummy Reference Algebra Just like spikes, dummy references, introduced by for example *reduction*, often get in the way in calculations. The Dummy Reference Algebra in Figure 8 provides rules for eliminating and moving dummy references.

5.4 Fixed-Point Induction

In this section we introduce the least fixed-point property for strong improvement, which will provide the principal tool for reasoning about the relative space behaviour of recursive functions, a simple form of fixed-point induction.

Space-faithful unwindings We start at the bottom. A consequence of Theorem 4.3 is that there is no bottom element in the space-ordering relation, since divergent terms containing different numbers of free variables are not cost equivalent – simply because when placed in a program context, their free variables can affect the amount of live data, and hence the space. The more free variables a divergent term contains the more space it can retain, and hence the lower in the improvement ordering it sits. This is significant when we define the notion of a chain of finite unwindings of a recursive definition. Usually the first approximation in such a chain is the bottom element but here we need to start from a divergent term with the right amount of free variables.

We are now ready to define precisely the space-faithful finite unwindings of a recursive definition.

Definition 5.9 (Finite Unwindings)
Let $\mathbb{V}$ be a value context with at least one occurrence of the hole. We define

$$
\begin{aligned}
{}^{w}R[^{v\curlyvee}M] &\lessapprox\gtrapprox {}^{w+v\curlyvee w}R[M] \\
{}^{w}R[^{v\curlywedge}M] &\lessapprox\gtrapprox {}^{v\curlywedge w}R[M] \\
\mathsf{let}\ \Gamma\ \mathsf{in}\ {}^{v\curlyvee}M &\lessapprox\gtrapprox {}^{v\curlyvee}\mathsf{let}\ \Gamma\ \mathsf{in}\ M \qquad \text{if } \operatorname{dom}\Gamma \subseteq \mathsf{FV}(M) \\
\mathsf{let}\ \Gamma\ \mathsf{in}\ {}^{v\curlywedge}M &\lessapprox\gtrapprox {}^{v+|\Gamma|\curlywedge}\mathsf{let}\ \Gamma\ \mathsf{in}\ M \qquad \text{if } \operatorname{dom}\Gamma \subseteq \mathsf{FV}(M) \\
{}^{w}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to {}^{w\curlyvee}N_i\} &\lessapprox\gtrapprox {}^{w}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\} \\
{}^{w\curlyvee v\curlyvee}M &\lessapprox\gtrapprox {}^{v\curlyvee}M \quad \text{if } w \leq v \\
{}^{w\curlywedge v\curlywedge}M &\lessapprox\gtrapprox {}^{v\curlywedge}M \quad \text{if } w \leq v \\
{}^{w\curlyvee X w\curlyvee}M &\lessapprox\gtrapprox {}^{w\curlyvee X}M \\
{}^{w\curlywedge X w\curlywedge}M &\lessapprox\gtrapprox {}^{w\curlywedge X}M \\
{}^{w\curlywedge v\curlyvee}M &\lessapprox\gtrapprox {}^{v\curlyvee w\curlywedge}M \\
{}^{w\curlyvee v\curlyvee}M &\lessapprox\gtrapprox {}^{v\curlyvee w\curlyvee}M \\
{}^{w\curlywedge v\curlywedge}M &\lessapprox\gtrapprox {}^{v\curlywedge w\curlywedge}M \\
{}^{w\curlyvee}M &\gtrapprox M \\
{}^{w\curlywedge}M &\gtrapprox M
\end{aligned}
$$

Figure 7: The Spike Algebra.

$$
\begin{aligned}
{}^{w}R[^{X}M] &\lessapprox\gtrapprox {}^{w\curlyvee X w}R[M] \\
\mathsf{let}\ \Gamma\ \mathsf{in}\ {}^{X}M &\lessapprox\gtrapprox {}^{|\Gamma|\curlywedge X}\mathsf{let}\ \Gamma\ \mathsf{in}\ M \quad \text{if } \operatorname{dom}\Gamma \subseteq \mathsf{FV}(M) \\
\mathsf{let}\ \Gamma\{{}^{w}_{0}x = V\}\ \mathsf{in}\ \mathbb{C}[^{\{x\}}M] &\lessapprox\gtrapprox \mathsf{let}\ \Gamma\{{}^{w}_{0}x = V\}\ \mathsf{in}\ \mathbb{C}[^{\mathsf{FV}(V)\setminus\{x\}}M] \\
{}^{w}R[^{X}M] &\lessapprox\gtrapprox {}^{w}R[M] \qquad \text{if } X \subseteq \mathsf{FV}(R) \\
{}^{\emptyset}M &\lessapprox\gtrapprox M \\
{}^{X\cup Y}M &\lessapprox\gtrapprox {}^{X}M \quad \text{if } Y \subseteq \mathsf{FV}(M) \\
{}^{XY}M &\lessapprox\gtrapprox {}^{X\cup Y}M \\
{}^{X}M &\gtrapprox M
\end{aligned}
$$

Figure 8: The Dummy Reference Algebra.

$\mathsf{let}\ \{^w_v f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^n]$ inductively by the following clauses.

$$\mathsf{let}\ \{^w_v f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^0] \stackrel{\text{def}}{=} \mathsf{let}\ \{^w_v f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[^{\{f\}}\Omega]$$
$$\mathsf{let}\ \{^w_v f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^{n+1}] \stackrel{\text{def}}{=} \mathsf{let}\ \{^w_v f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[^{w\curlyvee}\mathbb{V}[f^n]]$$

Syntactic Continuity Using the laws about Ω and the laws of unfolding it is easy to show that the approximations form an improvement chain: For all $0 \leq i < j$

$$\mathsf{let}\ \{^w_v f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^i] \gtrsim \mathsf{let}\ \{^w_v f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^j]$$

and that $\mathsf{let}\ \{^w_v f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f]$ is an upper bound of the chain - i.e., for all i, $\mathsf{let}\ \{^w_v f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^i] \gtrsim \mathsf{let}\ \{^w_v f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f]$.

The crucial property of strong improvement is that the relation is continuous with respect to unwinding of recursion. The definition of f is the *least* upper bound of this chain.

Theorem 5.10 (Syntactic Continuity)
$\mathsf{let}\ \{^w_v f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f] \gtrsim M \iff \forall n.\, \mathsf{let}\ \{^w_v f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^n] \gtrsim M$

This theorem forms the basis of the fixed-point induction technique which we spell out at the end of Section 5.5. In the rest of this section we develop the proof of Theorem 5.10. The reader less interested in the proof can safely move on to Section 5.5.

Before we proceed with the proof we need to introduce the following notation. We write $|\mathbb{C}| \geq n$ if all the holes in $\mathbb{C}$ are at a syntactic depth of at least n. So, for example, $|[\cdot]\, x| \geq 1$ and $|[\cdot]| \geq 0$. The following lemma captures the intuition that a computation of length n cannot depend on a subterm which is at a syntactic depth greater than n.

Lemma 5.11
If $\mathbb{C}[M]\Downarrow^n_{(h,s)}$, $|\mathbb{C}| > n$ and $\mathsf{FV}(N) \subseteq \mathsf{FV}(M)$ then

$$\mathbb{C}[N]\Downarrow^n_{(h,s)}.$$

Proof. The lemma is proved by induction over n. In every computation step we can go at most one level deeper in the term so we maintain the invariant that the term we want to replace is always at a deeper level than the length of the remaining computation. □

From Lemma 5.11 we can show that in a computation of length n we can safely replace a function call with its n'th unwinding:

Lemma 5.12
For every $\mathbb{C}'$, if $\mathbb{C}'[\mathsf{let}\ \{^w_v f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f]\sigma]\Downarrow^n_{(h,s)}$ then

$$\mathbb{C}'[\mathsf{let}\ \{^w_v f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^n]\sigma]\Downarrow^n_{(h,s)}$$

PROOF. Assume the premise. By repeatedly applying the unwinding rule for recursive definitions,

$$\mathsf{let}\ \{^{w}_{v} x = V\}\ \mathsf{in}\ \mathbb{C}[x] \lessapprox \mathsf{let}\ \{^{w}_{v} x = V\}\ \mathsf{in}\ \mathbb{C}[{}^{w\curlyvee}V] \qquad \text{if } x \in \mathsf{FV}(V)$$

we know that

$$\mathsf{let}\ \{^{w}_{v} f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f] \lessapprox \mathsf{let}\ \{^{w}_{v} f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[\mathbb{D}^n[f]]$$

where $\mathbb{D}$ is ${}^{w\curlyvee}\mathbb{V}[\cdot]$. Thus from the premise we have that

$$\mathbb{C}'[\mathsf{let}\ \{^{w}_{v} f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[\mathbb{D}^n[f]]\sigma]\Downarrow_{(h,s)}$$

and from the work on time improvements by Moran and Sands [MS99a] we know that it terminates in n steps. Note that $|\mathbb{C}'[\mathsf{let}\ \{^{w}_{v} f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[\mathbb{D}^n[\cdot]]\sigma]| > n$ and $\mathsf{FV}({}^{\{f\}}\Omega) \subseteq \mathsf{FV}(f)$ so by Lemma 5.11 we may replace f by ${}^{\{f\}}\Omega$:

$$\mathbb{C}'[\mathsf{let}\ \{^{w}_{v} f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[\mathbb{D}^n[{}^{\{f\}}\Omega]]\sigma] \equiv \mathbb{C}'[\mathsf{let}\ \{^{w}_{v} f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^n]\sigma]\Downarrow^{n}_{(h,s)}.$$

□

We are now ready to prove Theorem 5.10.

PROOF. The right-way implication follows from that for every n,

$$\mathsf{let}\ \{^{w}_{v} f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^n] \gtrsim \mathsf{let}\ \{^{w}_{v} f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f].$$

Consider the left-way implication. Assume the premise. We are required to show that $\mathsf{let}\ \{^{w}_{v} f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f] \gtrsim M$, that is, we need to show that for all $\mathbb{C}'$, σ such that $\mathbb{C}'[\mathsf{let}\ \{^{w}_{v} f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f]\sigma]$ and $\mathbb{C}'[M\sigma]$ are closed,

$$\mathbb{C}'[\mathsf{let}\ \{^{w}_{v} f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f]\sigma]\Downarrow_{(h,s)} \implies \mathbb{C}'[M\sigma]\Downarrow_{(h,s)}.$$

Thus assume $\mathbb{C}'[\mathsf{let}\ \{^{w}_{v} f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f]\sigma]\Downarrow_{(h,s)}$ in n steps. By Lemma 5.12 we may replace f by its n'th unwinding:

$$\mathbb{C}'[\mathsf{let}\ \{^{w}_{v} f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^n]\sigma]\Downarrow_{(h,s)}.$$

so from the main assumption we can conclude

$$\mathbb{C}'[M\sigma]\Downarrow_{(h,s)}$$

as required. □

5.5 Derivations in Context

We will often express properties which are relative to a fixed set of function definitions. It is cumbersome to carry such definitions in explicit let-terms, so we adopt a useful notation for derivations in context:

Definition 5.13
We write $\Gamma \vdash M \gtrsim N$ as an abbreviation for the following property: For all Γ', $\mathbb{C}$ and σ, if

- $\operatorname{dom}\Gamma' \cap \operatorname{dom}\Gamma = \emptyset$,
- $\mathsf{CV}(\mathbb{C}) \cap (\operatorname{dom}\Gamma \cup \mathsf{FV}(\Gamma)) = \emptyset$ and
- $\operatorname{dom}\sigma \cap (\operatorname{dom}\Gamma \cup \mathsf{FV}(\Gamma)) = \emptyset$,

then $\mathsf{let}\ \{\Gamma\Gamma'\}\ \mathsf{in}\ \mathbb{C}[M\sigma] \gtrsim \mathsf{let}\ \{\Gamma\Gamma'\}\ \mathsf{in}\ \mathbb{C}[N\sigma]$.

We will write a derivation

$$\Gamma \vdash M_0 \gtrsim M_1 \gtrsim M_2 \gtrsim \ldots$$

to mean $\Gamma \vdash M_0 \gtrsim M_1$ and $\Gamma \vdash M_1 \gtrsim M_2$ and so on. These contextual judgements satisfy a number of simple properties which facilitate their use.

Proposition 5.14
The following proof rules are sound:

- $$\frac{M \gtrsim N}{\Gamma \vdash M \gtrsim N}$$
- $$\frac{\Gamma \vdash M \gtrsim N \qquad \Gamma \vdash N \gtrsim L}{\Gamma \vdash M \gtrsim L}$$
- $$\frac{\Gamma \vdash M \gtrsim N \qquad \operatorname{dom}\Gamma' \cap \operatorname{dom}\Gamma = \emptyset}{\Gamma\Gamma' \vdash M \gtrsim N}$$
- $$\frac{\Gamma \vdash M \gtrsim N \qquad \mathsf{CV}(\mathbb{C}) \cap (\operatorname{dom}\Gamma \cup \mathsf{FV}(\Gamma)) = \emptyset}{\Gamma \vdash \mathbb{C}[M] \gtrsim \mathbb{C}[N]}$$
- $$\frac{\Gamma \vdash M \gtrsim N \qquad \operatorname{dom}\sigma \cap (\operatorname{dom}\Gamma \cup \mathsf{FV}(\Gamma)) = \emptyset}{\Gamma \vdash M\sigma \gtrsim N\sigma}$$
- $$\frac{\mathsf{FV}(V) \cap \mathsf{CV}(\mathbb{C}) = \emptyset}{\Gamma\{{}^0_0 f = V\} \vdash \mathbb{C}[f] \lessapprox \mathbb{C}[V]}$$

With the above notation and properties we have the following simple corollary of syntactic continuity, expressed in an informal natural-deduction style.

Corollary 5.15 (Fixed point induction)
The following proof rule is sound:

$$\frac{\Gamma \vdash \mathbb{C}[f^0] \gtrsim M \qquad \forall n \left(\begin{array}{c} \Gamma \vdash \mathbb{C}[f^n] \gtrsim M \\ \vdots \\ \Gamma \vdash \mathbb{C}[f^{n+1}] \gtrsim M \end{array} \right)}{\Gamma \vdash \mathbb{C}[f] \gtrsim M}$$

That is to say, if we can establish $\Gamma \vdash \mathbb{C}[f^0] \gtrsim M$ and that under the assumption that $\Gamma \vdash \mathbb{C}[f^n] \gtrsim M$ for some arbitrary n we can show $\Gamma \vdash \mathbb{C}[f^{n+1}] \gtrsim M$, then it holds that $\Gamma \vdash \mathbb{C}[f] \gtrsim M$.

6 Case Studies

Armed with a means to establish improvement properties for recursive functions, in the rest of this paper we will investigate the possibilities and limitations of space improvement.

The requirement is that transformed programs should improve on the space behaviour in all contexts. Are there any interesting transformations which are space improvements? In this section we present examples of some standard program transformations, and show how space improvement can be established using the tools from the previous sections. The results are not all positive; we will also show that there are many transformations that are not space improvements.

Case Study 1: Cyclic Structures

We will start with a very simple and intuitive space improvement which serves, above all else, to illustrate the use of the fixed-point induction method. We will show that the cyclic data structure $xs = x : xs$ improves on the non-cyclic structure that is generated by *repeat* x where *repeat* is defined as

$$repeat = \lambda x.\mathsf{let}\ \{ys = repeat\ x\}\ \mathsf{in}\ x : ys.$$

Using fixed-point induction we will prove a strong improvement property from which the desired weak improvement follows directly.

Proposition 6.1
Let Γ be $\{^0_0 repeat = \lambda x.\mathsf{let}\ \{ys = repeat\ x\}\ \mathsf{in}\ x : ys\}$ Then

$$\Gamma \vdash \mathsf{let}\ \{xs = repeat\ x\}\ \mathsf{in}\ M \gtrsim \mathsf{let}\ \{xs = x : xs\}\ \mathsf{in}\ M.$$

PROOF. We proceed by fixed-point induction over the definition of *repeat*. The following derivation shows the base case.

$$
\begin{array}{l}
\Gamma \vdash \mathsf{let}\ \{xs = \mathit{repeat}^0\, x\}\ \mathsf{in}\ M \\
\quad \equiv (\textit{definition of unwindings}) \\
\qquad \mathsf{let}\ \{xs = {}^{\{\mathit{repeat}\}}\Omega\, x\}\ \mathsf{in}\ M \\
\quad \gtrsim (\textit{the dummy reference algebra}) \\
\qquad \mathsf{let}\ \{xs = \Omega\, x\}\ \mathsf{in}\ M \\
\quad \lesssim (\textit{rules for}\ \Omega) \\
\qquad \mathsf{let}\ \{xs = {}^{\{x\}}\Omega\}\ \mathsf{in}\ ) \\
\quad \gtrsim (\textit{rules for}\ \Omega) \\
\qquad \mathsf{let}\ \{xs = x : xs\}\ \mathsf{in}\ M
\end{array}
$$

And this derivation shows the inductive step.

$$
\begin{array}{l}
\Gamma \vdash \mathsf{let}\ \{xs = \mathit{repeat}^{n+1}\, x\}\ \mathsf{in}\ M \\
\quad \equiv (\textit{definition of unwindings}) \\
\qquad \mathsf{let}\ \{xs = {}^{0\curlyvee}(\lambda x.\mathsf{let}\ \{ys = \mathit{repeat}^n\, x\}\ \mathsf{in}\ x : ys)\, x\}\ \mathsf{in}\ M \\
\quad \lesssim (\textit{the spike algebra}) \\
\qquad \mathsf{let}\ \{xs = (\lambda x.\mathsf{let}\ \{ys = \mathit{repeat}^n\, x\}\ \mathsf{in}\ x : ys)\, x\}\ \mathsf{in}\ M \\
\quad \lesssim (\textit{reduction}) \\
\qquad \mathsf{let}\ \{xs = {}^{\curlyvee}\mathsf{let}\ \{ys = \mathit{repeat}^n\, x\}\ \mathsf{in}\ x : ys\}\ \mathsf{in}\ M \\
\quad \gtrsim (\textit{the spike algebra}) \\
\qquad \mathsf{let}\ \{xs = \mathsf{let}\ \{ys = \mathit{repeat}^n\, x\}\ \mathsf{in}\ x : ys\}\ \mathsf{in}\ M \\
\quad \gtrsim (\textit{ih}) \\
\qquad \mathsf{let}\ \{xs = \mathsf{let}\ \{ys = x : ys\}\ \mathsf{in}\ x : ys\}\ \mathsf{in}\ M \\
\quad \gtrsim (\textit{value-merge'}) \\
\qquad \mathsf{let}\ \{xs = x : xs\}\ \mathsf{in}\ M
\end{array}
$$

□

Case Study 2: Intermediate Data Structures

Our next example concerns intermediate data structures produced by a definition of the Haskell prelude function *any*.[6] The function takes two arguments: a

[6] The example and its space properties were discussed on the Haskell mailing list in January 2001 (www.haskell.org).

predicate p and a list xs and tests whether any of the elements of the list fulfils the predicate. The function can be defined in a direct recursive style:

$$\begin{aligned} any\ p\ xs = \mathsf{case}\ xs\ \mathsf{of} \\ \qquad \mathsf{nil} \to false \\ \qquad y : ys \to p\ y \mid\mid any\ p\ ys \end{aligned}$$

where $||$ is the infix logical *or* operator. However in the Haskell report [PJHA+99] *any* is defined in an elegant combinator style:

$$any'\ p = or\ \circ\ map\ p$$

where *or* is defined as

$$or = foldr\ (||)\ false$$

and *map*, *foldr* and *false* are top-level definitions defined as usual. Apart from the stylistic differences, there is a key operational difference between the two definitions. The latter, when applied to p and xs, builds a list $map\ p\ xs$. Interestingly, several discussions on the Haskell mailing list where concerned about the efficiency of the latter definition. In particular, that the construction of the list would lead to a space leak proportional to the length of the list. The replies on the mailing list were of two kinds. The first kind emphasised that the definition in the Haskell report should be seen as a specification (a reference implementation) of only the extensional behaviour of *any*. A particular code distribution would be free to provide the presumably more efficient definition of *any*. A clever compiler might even automatically derive it using *deforestation* [Wad90]. The second kind of reply appealed to the folklore of call-by-need: the list is only an intermediate data structures and the two definitions have the same asymptotic space behaviour. The following result confirms the folklore.

Proposition 6.2
$\Gamma \vdash any\ p\ xs \underset{\approx}{\Leftrightarrow} any'\ p\ xs$
where Γ contains the definitions of *any* and *any'* (and the definitions of the other functions they rely on).

The relevance of the result is twofold. Firstly, the definition in the Haskell report is at most a constant factor worse than the direct recursive definition so it serves perfectly well as a reference implementation with respect to space use. Secondly, a compiler which replaces the latter definition with the former doesn't risk to introduce a space leak in some weird case. This might seem obvious at first thought but having worked with space improvement for a while we have learnt to not jump to such conclusions. See case study 3, 5 and 6 for some examples that failed in contrast to our initial intuition.

Let us sketch the proof of Proposition 6.2. As you would expect the proof is via a strong improvement. However the proof is considerably more involved

than our previous example because we cannot show the strong improvement

$$\Gamma \vdash \mathit{any}\, p\, xs \mathrel{\lhd\!\!\!\sim\!\!\!\rhd} \mathit{any}'\, p\, xs$$

because *any* and *any′* uses different amounts of space – although the difference is within a constant factor. The solution is to introduce alternative definitions of *any* and *any′* which we will call any_a and any'_a respectively such that

$$\Gamma \vdash \mathit{any}\, p\, xs \mathrel{\lhd\!\!\!\approx\!\!\!\rhd} \mathit{any}_a \cdot p \cdot xs \mathrel{\lhd\!\!\!\sim\!\!\!\rhd} \mathit{any}'_a \cdot p \cdot xs \mathrel{\lhd\!\!\!\approx\!\!\!\rhd} \mathit{any}'\, p\, xs.$$

To come up with the definitions of any_a and any'_a is non trivial and requires some creativity and/or hard work. Our experience has led us to the following methodology: We modify the original definitions in a way such that

- the modified definitions are weakly equivalent to the original definitions, and
- we can show it just using the laws of weak improvement without the need of fixed-point induction.

The modifications are of two kinds:

- First, wherever it can be justified, we put in zero weights on short-lived structures such as arguments to known functions. This reduces the "noise" from the computations and is sometimes necessary to make the definitions strongly equivalent. It also vastly simplifies the proof of the strong improvement since it eliminates lots of spikes that would otherwise clutter the derivations.
- Second, more difficult step: whenever the two original definitions have a different space behaviour modulo "noise", we level them up by adding spikes, dummy bindings and extra weights. However when we do this we have to be careful to not increase space use by more than a constant factor.

Let us return to our example. In the first step we add zero weight on all applications of known functions. We make these modifications also to *foldr*, *map* and *or* which are called by *any′*. We call the modified definitions foldr_a, map_a and or_a respectively. The second step is to add dummy space use to *any* to make it take up as much space as *any′*. Recall the definition of *any*:

$$\begin{aligned}\mathit{any} = \lambda p.\lambda xs.&\mathsf{case}\ xs\ \mathsf{of}\\ &\mathsf{nil} \to \mathsf{false}\\ &y : ys \to (||)\ (p\ y)\ (\mathit{any}\ p\ ys)\end{aligned}$$

We modify the definition as follows.

$$\begin{aligned}\mathit{any}_a = \lambda p.\lambda xs.{}^{\lambda 2}&\mathsf{case}\ xs\ \mathsf{of}\\ &\mathsf{nil} \to {}^{\curlyvee}\mathsf{false}\\ &y : ys \to {}^{\curlyvee}\mathsf{let}\ z = \Omega\\ &\qquad\qquad \mathsf{in}\ (||) \cdot (p \cdot y) \cdot ({}^{\{z\}}(\mathit{any} \cdot p \cdot ys))\end{aligned}$$

There are two interesting modifications. The first one is the extra weight on the case expression, which compensates for the extra stack space used by any'; for any' to scrutinise the head of its input xs it calls or with the argument $map\,p\,xs$ and or passes the argument to $foldr$. Then, $foldr$ pushes a stack-frame and forces the computation of its input $map\,p\,xs$. In turn, map pushes a stack-frame and forces the computation of xs. Thus two stack-frames have been pushed onto the stack. The extra weight on the case in any_a mimics this behaviour. From Lemma 5.6 we know that the extra weight can make any_a use up at most a constant factor more stack than any.

The other interesting modification is the dummy binding of z in the cons-branch of any_a. The dummy binding lives until $any \cdot p \cdot ys$ or if it never happens when the closure that holds $any \cdot p \cdot ys$ becomes garbage. We get this effect because of the dummy reference to z. The dummy binding is there to mimic the space used up by the list $map\,p\,xs$ which any' constructs. It is worth noting that although the list is an intermediate data structure it is not necessarily short-lived. It will stay in memory during the evaluation of $p\,y$ which can be arbitrarily long and which may even call any' itself. But the extra structure can not change the asymptotic space behaviour because there are other structures in the heap which are at least as long lived. This is not easy to see from the definition of any' but if we spell out the defintion of any_a without syntactic sugar

$$
\begin{array}{l}
any_a = \lambda p.\lambda xs.{}^{\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\qquad\qquad \mathsf{nil} \to {}^{\curlyvee}\mathsf{false} \\
\qquad\qquad y : ys \to {}^{\curlyvee}\mathsf{let}\ z = \Omega \\
\qquad\qquad\qquad\qquad\quad a = p \cdot y \\
\qquad\qquad\qquad\qquad\quad b = {}^{\{z\}}\, any_a \cdot p \cdot ys \\
\qquad\qquad\qquad\qquad \mathsf{in}\ (||) \cdot a \cdot b
\end{array}
$$

we can see that the dummy binding that mimics the structure cannot live longer than the binding b (Lemma 5.3). With these appropriate definitions of any_a and any'_a it is straightforward to show

$$\Gamma \vdash any\,p\,xs \mathrel{\overset{\diamond}{\approx}} any_a \cdot p \cdot xs$$

and

$$\Gamma \vdash any'_a \cdot p \cdot xs \mathrel{\overset{\diamond}{\approx}} any'\,p\,xs.$$

as outlined above. It remains to show that

$$\Gamma \vdash any_a \cdot p \cdot xs \mathrel{\overset{\diamond}{\lesssim}} any'_a \cdot p \cdot xs$$

which requires a substantial effort although it is quite straightforward as we will show below.

PROOF. Since any'_a is not directly recursively defined we first note that

$$\Gamma \vdash any'_a \cdot p \cdot xs \mathrel{\lhd\!\rhd}_{\sim} foldr_a \cdot (||) \cdot false \cdot (map_a \cdot p \cdot xs)$$

so we can reason via the finite unwindings of $foldr_a$. We will show by induction over n that for every n,

$$any^n_a \cdot p \cdot xs \mathrel{\lhd\!\rhd}_{\sim} foldr^n_a \cdot (||) \cdot false \cdot (map_a \cdot p \cdot xs)$$

and the result then follows by continuity. In the base case we need to show that

$$\Gamma \vdash {}^{\{any_a\}}\Omega \cdot p \cdot xs \mathrel{\lhd\!\rhd}_{\sim} {}^{\{foldr_a\}}\Omega \cdot (||) \cdot false \cdot (map_a \cdot p \cdot xs).$$

The derivation is straightforward and has been omitted. We just want to note that the strong improvement holds only because the bindings for the top-level functions in Γ have zero weight so a reference to a top-level function can't hold on to any space. Now let us consider the inductive step. We are required to show that

$$any^{n+1}_a \cdot p \cdot xs \mathrel{\lhd\!\rhd}_{\sim} foldr^{n+1}_a \cdot (||) \cdot false \cdot (map_a \cdot p \cdot xs)$$

under the assumption that

$$any^n_a \cdot p \cdot xs \mathrel{\lhd\!\rhd}_{\sim} foldr^n_a \cdot (||) \cdot false \cdot (map_a \cdot p \cdot xs).$$

Rather than starting from $any^{n+1}_a \cdot p \cdot xs$ and deriving $foldr^{n+1}_a \cdot (||) \cdot false \cdot (map_a \cdot p \cdot xs)$ we do the opposite. We have found that it is often easiest to start with the definition with the least number of "space gadgets" which usually is the definition that without the gadgets would require most space. In this way it is often possible to derive which gadgets are required in the cheaper definition to make the definitions strongly space equivalent. The derivation is as follows.

$$\Gamma \vdash foldr^{n+1}_a \cdot (||) \cdot false \cdot \big(map_a \cdot p \cdot xs\big)$$

As a first step we eliminate the syntactic sugar in the application so that we can apply the rules for reduction:

$$\begin{array}{l}
\quad \equiv \\
\quad \mathsf{let}\ ys = map_a \cdot p \cdot xs \\
\quad \mathsf{in}\ foldr^{n+1}_a \cdot (||) \cdot false \cdot ys \\
\mathrel{\lhd\!\rhd}_{\sim} (\textit{reduction etc.}) \\
\quad \mathsf{let}\ ys = map_a \cdot p \cdot xs \\
\quad \mathsf{in\ case}\ ys\ \mathsf{of} \\
\qquad \textit{branches of}\ foldr^{n+1}_a \sigma_0
\end{array}$$

Note that, because we have put zero weights on the applications the reduction leaves no spikes behind. The next step is to float the let binding to its use and eliminate it:

$$
\begin{array}{l}
\lesssim (\textit{let-R}) \\
\quad \mathsf{case}\ \begin{pmatrix} \mathsf{let}\ ys = map_a \cdot p \cdot xs \\ \mathsf{in}\ ys \end{pmatrix}\ \mathsf{of} \\
\quad \textit{branches of } foldr_a^{n+1}\sigma_0 \\
\lesssim (\textit{let-elim}) \\
\quad \mathsf{case}\ {}^{1\curlywedge}\left(map_a \cdot p \cdot xs\right)\ \mathsf{of} \\
\quad \textit{branches of } foldr_a^{n+1}\sigma_0
\end{array}
$$

The elimination of the let leaves a heap spike behind. Spikes are a nuisance and therefore we try to eliminate them by attaching zero weights whereever possible. At this point it may seem as if we could have put a zero heap weight on the binding we eliminated because it is clearly short lived. But if we do so it turns out that the induction hypothesis is not of the right form. In this case the spike poses little problem and we just float it away and then reduce the call to *map*:

$$
\begin{array}{l}
\lesssim (\textit{spike algebra}) \\
\quad {}^{1\curlywedge}\mathsf{case}\ map_a \cdot p \cdot xs\ \mathsf{of} \\
\quad\quad \textit{branches of } foldr_a^{n+1}\sigma_0 \\
\lesssim (\textit{reduction etc.}) \\
\quad {}^{1\curlywedge}\mathsf{case}\ \begin{pmatrix} \mathsf{case}\ xs\ \mathsf{of} \\ \quad \mathsf{nil} \to \textit{nil-branch of } map_a\sigma_1 \\ \quad b : bs \to \textit{cons-branch of } map_a\sigma_1 \end{pmatrix}\ \mathsf{of} \\
\quad\quad \textit{branches of } foldr_a^{n+1}\sigma_0
\end{array}
$$

Next, we apply the case of case transformation step. Note that it leads to an extra weight on the case expression which explains the extra weight in the definition of any_a. This is an example of how we may derive the gadgets we need in the gadget versions.

$$
\begin{array}{l}
\lesssim (\textit{R-case}) \\
\quad {}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\qquad \mathsf{nil} \to \mathsf{case}\ \textit{nil-branch of } map_a\sigma_1\ \mathsf{of} \\
\qquad\quad \textit{branches of } foldr_a^{n+1}\sigma_0 \\
\qquad b : bs \to \mathsf{case}\ \textit{cons-branch of } map_a\sigma_1\ \mathsf{of} \\
\qquad\quad\quad \textit{branches of } foldr_a^{n+1}\sigma_0
\end{array}
$$

We proceed with the derivation in the nil-branch:

$$
\begin{array}{l}
\quad \equiv \\
{}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\quad \mathsf{nil} \to \mathsf{case}\ \mathsf{nil}\ \mathsf{of} \\
\qquad\quad \mathit{branches\ of}\ \mathit{foldr}_a^{n+1}\sigma_0 \\
\quad b : bs \to \mathsf{case}\ \mathit{cons\text{-}branch\ of}\ \mathit{map}_a\sigma_1\ \mathsf{of} \\
\qquad\quad \mathit{branches\ of}\ \mathit{foldr}_a^{n+1}\sigma_0 \\
\lessapprox\gtrapprox (\mathit{reduction}) \\
{}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\quad \mathsf{nil} \to {}^{1\curlyvee\{(|\,|),\mathit{foldr}_a\}}\mathit{false} \\
\quad b : bs \to \mathsf{case}\ \mathit{cons\text{-}branch\ of}\ \mathit{map}_a\sigma_1\ \mathsf{of} \\
\qquad\quad \mathit{branches\ of}\ \mathit{foldr}_a^{n+1}\sigma_0
\end{array}
$$

Note how the reduction leaves dummy references behind. The references occured in the cons-branch which was discarded. However they refer to zero weight top-level defintions so we may discard them:

$$
\begin{array}{l}
\lessapprox\gtrapprox (\mathit{dummy\ reference\ algebra}) \\
{}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\quad \mathsf{nil} \to {}^{1\curlyvee}\mathit{false} \\
\quad b : bs \to \mathsf{case}\ \mathit{cons\text{-}branch\ of}\ \mathit{map}_a\sigma_1\ \mathsf{of} \\
\qquad\quad \mathit{branches\ of}\ \mathit{foldr}_a^{n+1}\sigma_0 \\
\quad \equiv \\
{}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\quad \mathsf{nil} \to \mathit{nil\text{-}branch\ of}\ \mathit{any}_a^{n+1}\sigma_2 \\
\quad b : bs \to \mathsf{case}\ \mathit{cons\text{-}branch\ of}\ \mathit{map}_a\sigma_1\ \mathsf{of} \\
\qquad\quad \mathit{branches\ of}\ \mathit{foldr}_a^{n+1}\sigma_0
\end{array}
$$

We proceed with the derivation in the cons-branch.

$$
\begin{array}{l}
\quad \equiv \\
{}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\quad \mathsf{nil} \to \mathit{nil\text{-}branch\ of}\ \mathit{any}_a^{n+1}\sigma_2 \\
\quad b : bs \to \mathsf{case}\ \left(p \cdot b\right) : \left(\mathit{map}_a \cdot p \cdot bs\right)\ \mathsf{of} \\
\qquad\quad \mathit{branches\ of}\ \mathit{foldr}_a^{n+1}\sigma_0
\end{array}
$$

At this point it may be tempting to apply a rule for reduction to reduce the case expression. But it is absolutely crucial to note that we are using syntactic sugar in the constructor application. Our laws can only be applied to terms without syntactic sugar. Indeed the constructor application is not even a value but syntactic sugar for a let-expression:

$$
\begin{array}{l}
\equiv \\
{}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\quad \mathsf{nil} \to \mathit{nil\text{-}branch}\ \mathit{of}\ \ \mathit{any}_a^{n+1}\sigma_2 \\
\quad b : bs \to \mathsf{case} \begin{pmatrix} \mathsf{let}\ c = p \cdot b \\ \quad\ cs = \mathit{map}_a \cdot p \cdot bs \\ \mathsf{in}\ c : cs \end{pmatrix} \mathsf{of} \\
\qquad\qquad \mathit{branches}\ \mathit{of}\ \mathit{foldr}_a^{n+1}\sigma_0
\end{array}
$$

It is now apperent that we must float the lets away before we can perform the reduction.

$$
\begin{array}{l}
\lessapprox\gtrapprox(\mathit{let\text{-}R}) \\
\quad {}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\qquad \mathsf{nil} \to \mathit{nil\text{-}branch}\ \mathit{of}\ \ \mathit{any}_a^{n+1}\sigma_2 \\
\qquad b : bs \to \mathsf{let}\ c = p \cdot b \\
\qquad\qquad\qquad\ \ cs = \mathit{map}_a \cdot p \cdot bs \\
\qquad\qquad\ \ \mathsf{in\ case}\ c : cs\ \mathsf{of} \\
\qquad\qquad\qquad \mathit{branches}\ \mathit{of}\ \mathit{foldr}_a^{n+1}\sigma_0 \\
\lessapprox\gtrapprox(\mathit{reduction}) \\
\quad {}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\qquad \mathsf{nil} \to \mathit{nil\text{-}branch}\ \mathit{of}\ \ \mathit{any}_a^{n+1}\sigma_2 \\
\qquad b : bs \to \mathsf{let}\ c = p \cdot b \\
\qquad\qquad\qquad\ \ cs = \mathit{map}_a \cdot p \cdot bs \\
\qquad\qquad\ \ \mathsf{in}\ {}^{1\curlyvee}\mathsf{let}\ ds = \mathit{foldr}_a^n \cdot (||) \cdot \mathit{false} \cdot cs \\
\qquad\qquad\qquad\ \ \mathsf{in}\ (||) \cdot c \cdot ds
\end{array}
$$

We are now at a point when we have performed all "natural" transformation steps. To figure out what to do next it is helpful to recall the induction hypothesis: $any_a^n \cdot p \cdot xs \mathrel{\lhd\!\!\sim} foldr_a^n \cdot (||) \cdot false \cdot (map_a \cdot p \cdot xs)$. To be able to apply the induction hypotheis we need to get the term in this form. We start by moving away the spike and then we float the lets around:

$\lhd\!\!\sim$(*spike algebra*)

$$
\begin{array}{l}
{}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\quad \mathsf{nil} \to \mathit{nil\text{-}branch\ of}\ \ any_a^{n+1}\sigma_2 \\
\quad b : bs \to {}^{1\curlyvee}\mathsf{let}\ c = p \cdot b \\
\qquad\qquad\qquad\qquad cs = map_a \cdot p \cdot bs \\
\qquad\qquad\qquad\quad \mathsf{in\ let}\ ds = foldr_a^n \cdot (||) \cdot false \cdot cs \\
\qquad\qquad\qquad\qquad \mathsf{in}\ (||) \cdot c \cdot ds
\end{array}
$$

$\lhd\!\!\sim$(*let-flatten*)

$$
\begin{array}{l}
{}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\quad \mathsf{nil} \to \mathit{nil\text{-}branch\ of}\ \ any_a^{n+1}\sigma_2 \\
\quad b : bs \to {}^{1\curlyvee}\mathsf{let}\ c = p \cdot b \\
\qquad\qquad\qquad\qquad cs = map_a \cdot p \cdot bs \\
\qquad\qquad\qquad\qquad ds = foldr_a^n \cdot (||) \cdot false \cdot cs \\
\qquad\qquad\qquad\quad \mathsf{in}\ (||) \cdot c \cdot ds
\end{array}
$$

$\lhd\!\!\sim$(*let-flatten*)

$$
\begin{array}{l}
{}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\quad \mathsf{nil} \to \mathit{nil\text{-}branch\ of}\ \ any_a^{n+1}\sigma_2 \\
\quad b : bs \to {}^{1\curlyvee}\mathsf{let}\ cs = map_a \cdot p \cdot bs \\
\qquad\qquad\qquad\qquad ds = foldr_a^n \cdot (||) \cdot false \cdot cs \\
\qquad\qquad\qquad\quad \mathsf{in\ let}\ c = p \cdot b \\
\qquad\qquad\qquad\qquad \mathsf{in}\ (||) \cdot c \cdot ds
\end{array}
$$

We are now about to float the binding for cs into the binding for ds. This may lead to arbitrarily delayed allocation so we need to compensate by a dummy binding.

$\lhd\!\!\sim$(*let-let'*)

$$
\begin{array}{l}
{}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\quad \mathsf{nil} \to \mathit{nil\text{-}branch\ of}\ \ any_a^{n+1}\sigma_2 \\
\quad b : bs \to {}^{1\curlyvee}\mathsf{let}\ z = \Omega \\
\qquad\qquad\qquad\qquad ds = {}^{\{z\}}\mathsf{let}\ cs = map_a \cdot p \cdot bs \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ foldr_a^n \cdot (||) \cdot false \cdot cs \\
\qquad\qquad\qquad\quad \mathsf{in\ let}\ c = p \cdot b \\
\qquad\qquad\qquad\qquad \mathsf{in}\ (||) \cdot c \cdot ds
\end{array}
$$

We introduce syntactic sugar:

$$
\begin{array}{l}
\equiv \\
{}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\quad \mathsf{nil} \to \mathit{nil\text{-}branch}\ \mathit{of}\ \ \mathit{any}_a^{n+1}\sigma_2 \\
\quad b : bs \to {}^{1\curlyvee}\mathsf{let}\ z = \Omega \\
\qquad\qquad\qquad ds = {}^{\{z\}}\mathit{foldr}_a^n \cdot (||) \cdot \mathit{false} \cdot \left(\mathit{map}_a \cdot p \cdot bs\right) \\
\qquad\qquad \mathsf{in}\ \mathsf{let}\ c = p \cdot b \\
\qquad\qquad\quad \mathsf{in}\ (||) \cdot c \cdot ds
\end{array}
$$

We are finally at the point where we can apply the induction hypothesis:

$$
\begin{array}{l}
\lessapprox \\
{}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\quad \mathsf{nil} \to \mathit{nil\text{-}branch}\ \mathit{of}\ \ \mathit{any}_a^{n+1}\sigma_2 \\
\quad b : bs \to {}^{1\curlyvee}\mathsf{let}\ z = \Omega \\
\qquad\qquad\qquad ds = {}^{\{z\}}\left(\mathit{any}_a^n \cdot p \cdot bs\right) \\
\qquad\qquad \mathsf{in}\ \mathsf{let}\ c = p \cdot b \\
\qquad\qquad\quad \mathsf{in}\ (||) \cdot c \cdot ds
\end{array}
$$

It only remains to wrap things up. We need to float the lets to the right places so we can introduce the syntactic sugar:

$$
\begin{array}{l}
\lessapprox (\mathit{let\text{-}R}) \\
{}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\quad \mathsf{nil} \to \mathit{nil\text{-}branch}\ \mathit{of}\ \ \mathit{any}_a^{n+1}\sigma_2 \\
\quad b : bs \to {}^{1\curlyvee}\mathsf{let}\ z = \Omega \\
\qquad\qquad\qquad ds = {}^{\{z\}}\left(\mathit{any}_a^n \cdot p \cdot bs\right) \\
\qquad\qquad \mathsf{in}\ \begin{pmatrix}\mathsf{let}\ c = p \cdot b \\ \mathsf{in}\ (||) \cdot c\end{pmatrix} \cdot ds \\
\quad \equiv \\
{}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\quad \mathsf{nil} \to \mathit{nil\text{-}branch}\ \mathit{of}\ \ \mathit{any}_a^{n+1}\sigma_2 \\
\quad b : bs \to {}^{1\curlyvee}\mathsf{let}\ z = \Omega \\
\qquad\qquad\qquad ds = {}^{\{z\}}\left(\mathit{any}_a^n \cdot p \cdot bs\right) \\
\qquad\qquad \mathsf{in}\ (||) \cdot \left(p \cdot b\right) \cdot ds \\
\lessapprox (\mathit{let\text{-}flatten}) \\
{}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\quad \mathsf{nil} \to \mathit{nil\text{-}branch}\ \mathit{of}\ \ \mathit{any}_a^{n+1}\sigma_2 \\
\quad b : bs \to {}^{1\curlyvee}\mathsf{let}\ z = \Omega \\
\qquad\qquad \mathsf{in}\ \mathsf{let}\ ds = {}^{\{z\}}\left(\mathit{any}_a^n \cdot p \cdot bs\right) \\
\qquad\qquad\quad \mathsf{in}\ (||) \cdot \left(p \cdot b\right) \cdot ds
\end{array}
$$

$$
\begin{array}{l}
\equiv \\
{}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\quad \mathsf{nil} \to \mathit{nil\text{-}branch\ of}\ \ \mathit{any}_a^{n+1}\sigma_2 \\
\quad b : bs \to {}^{1\curlyvee}\mathsf{let}\ z = \Omega \\
\qquad\qquad \mathsf{in}\ (||) \cdot \big(p \cdot b\big) \cdot {}^{\{z\}}\big(\mathit{any}_a^n \cdot p \cdot bs\big)
\end{array}
$$

We have now got to the definition of any_a:

$$
\begin{array}{l}
\equiv \\
{}^{1\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\
\quad \mathsf{nil} \to \mathit{nil\text{-}branch\ of}\ \ \mathit{any}_a^{n+1}\sigma_2 \\
\quad b : bs \to \mathit{cons\text{-}branch\ of}\ \ \mathit{any}_a^{n+1}\sigma_2
\end{array}
$$

so it only remains to expand the redexes.

$$
\begin{array}{l}
\lessapprox (\mathit{reduction\ etc.}) \\
\mathit{any}_a^{n+1} \cdot p \cdot xs
\end{array}
$$

□

The plethora of spikes, dummy references, dummy bindings and weights that are necessary in the derivations make the process of constructing derivations like this one extremely error prone. We found it necessary to develop a simple tool to formally check derivations, and the steps of this derivation have been verified in this way.

Case Study 3: Trading Stack for Heap

This case study is about the associativity of append, $(+\!\!+)$. It is interesting because it is an example of a transformation that can increase heap usage with more than a constant factor so it falls outside of $\lessapprox$. However the transformation can only lead to a constant factor difference in the *total amount* of space used. The reason is that in all cases where the amount of heap increases, a corresponding amount of stack space is used already.

To make this claim precise we define a relaxed version of $\gtrsim$, which allows stack space to be traded for heap space:

Definition 6.3 (Stack Weak Improvement)
We say that M *is stack weakly improved by* N, written $M \blacktriangleright\!\!\gtrsim N$, if there exists a linear function $f \in \mathbb{N} \to \mathbb{N}$ such that for all $\mathbb{C}$, σ such that $\mathbb{C}[M\sigma]$ and $\mathbb{C}[N\sigma]$ are closed,

$$\mathbb{C}[M\sigma]\Downarrow_{(h,s)} \implies \mathbb{C}[N\sigma]\Downarrow_{(h',s')}.$$

for some h' and s' such that $s' \leq f(s)$ and $h' + s' \leq f(h+s)$.

We can now state an improvement property of append:

Proposition 6.4
$(xs +\!\!+ ys) +\!\!+ zs \mathrel{\lhd\!\!\approx} xs +\!\!+ (ys +\!\!+ zs)$.

Note that the relaxed relation is only required in one direction. But it is the direction that one most often would like to use when applying this equivalence – it can lead to an asymptotic speedup in some contexts. We will see such an example later.

Now let us outline the proof of Proposition 6.4. We will follow the methodology from the previous example and come up with modified versions of append for which we can establish a strong improvement. We will need four different versions, one for each occurrence of append, which we call $+\!\!+_a, \ldots, +\!\!+_d$. The strong improvement part of the proposition turns out to be valuable in its own right (see case study 4) so we spell it out here.

Lemma 6.5

$$\Gamma \vdash \mathsf{let}\ \{ps = (+\!\!+_a) \cdot xs \cdot ys\}\ \mathsf{in}\ (+\!\!+_b) \cdot ps \cdot zs \mathrel{\lhd\!\!\approx} \mathsf{let}\ \{qs = (+\!\!+_d) \cdot ys \cdot zs\}\ \mathsf{in}\ (+\!\!+_c) \cdot xs \cdot qs$$

We have stated the lemma without syntactic sugar. We have found that this is often the first step towards an intuition about space use. Indeed, it is now explicit that the terms allocate space in the heap before they call the append function. How long lived are these bindings? Clearly, the binding for ps in the left hand side of the improvement is very short lived: $+\!\!+_b$ immediately evaluates its first argument and then there is no remaining references to ps. However in the right hand side the binding for qs may live for a long time. To compensate for this and make the strong improvement hold we have added a dummy allocation in the definition of $+\!\!+_a$:

$$\begin{array}{l}(+\!\!+_a) = \lambda as.\lambda bs.\mathsf{let}\ z = \Omega \\ \qquad\qquad \mathsf{in\ case}\ as\ \mathsf{of} \\ \qquad\qquad\quad \mathsf{nil} \to {}^{\{z\}\wedge} bs \\ \qquad\qquad\quad c : cs \to {}^{\{z\}}\mathsf{let}\ ds = (+\!\!+_a) \cdot cs \cdot bs \\ \qquad\qquad\qquad\qquad \mathsf{in}\ c : ds\end{array}$$

The dummy binding is allocated just before the case expression is executed, and lives until just after a branch has been selected. Thus the lifetime of the binding matches the lifetime of the stackframe pushed for the case expression. The binding exactly compensates for the different heap behaviours of the original functions. There is also a difference in stack usage between $(xs +\!\!+ ys) +\!\!+ zs$ and $xs +\!\!+ (ys +\!\!+ zs)$. This difference is of a similar nature to the difference between

any and *any'* from our previous case study. We need to put an extra weight on the case in $+\!\!+_c$:

$$\begin{array}{l}(+\!\!+_c) = \lambda as.\lambda bs.{}^2\mathsf{case}\ as\ \mathsf{of}\\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{nil} \to bs\\ \qquad\qquad\qquad\qquad\qquad\qquad c : cs \to \mathsf{let}\ ds = (+\!\!+_c) \cdot cs \cdot bs\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ c : ds\end{array}$$

For $+\!\!+_b$ and $+\!\!+_d$ the modifications are minor and only involves zero weights on short lived stack elements. With these definitions at hand it is not difficult to show Lemma 6.5 although the derivations are lengthy.

It easy to see that the modifications in $+\!\!+_c$ and $+\!\!+_d$ are within a constant factor of the original definition of append (Lemma 5.6 etc.), so we have

$$\Gamma \vdash \mathsf{let}\ \{qs = (+\!\!+_d) \cdot ys \cdot zs\}\ \mathsf{in}\ (+\!\!+_c) \cdot xs \cdot qs \mathrel{\lessapprox\!\!\gtrapprox} xs +\!\!+ (ys +\!\!+ zs).$$

To show Proposition 6.4 it remains to show that

$$\Gamma \vdash (xs +\!\!+ ys) +\!\!+ zs \mathrel{\lessapprox\!\!\gtrapprox} \mathsf{let}\ \{ps = (+\!\!+_a) \cdot xs \cdot ys\}\ \mathsf{in}\ (+\!\!+_b) \cdot as \cdot zs.$$

The difficulty lies in the dummy binding in the definition of $+\!\!+_a$. Recall that the lifetime of the dummy binding precisely matches the lifetime of the stack frame pushed by the case. Such a binding can at most double the *total amount* of space use – hence it is within a constant factor as stated by this lemma.

Lemma 6.6

$$\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\} \mathrel{\lessapprox\!\!\gtrapprox} \mathsf{let}\ \{z = \Omega\}\ \mathsf{in}\ \mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to {}^{\{z\}}N_i\}$$

This completes the proof sketch of Proposition 6.4.

In the beginning of this section we made another claim which partly motivated the introduction of a new relation, namely that the transformation can lead to an asymptotic increase in heap usage. The following family of contexts, indexed by k shows that $\Gamma \vdash (xs +\!\!+ ys) +\!\!+ zs \not\mathrel{\lessapprox\!\!\gtrapprox} xs +\!\!+ (ys +\!\!+ zs)$ by exhibiting a difference in heap behaviour which grows with k.

$$\begin{array}{l}\mathsf{let}\ g\ k\ ys\ zs = \mathsf{if}\ k = 0\\ \qquad\qquad\qquad\qquad\quad \mathsf{then\ nil}\\ \qquad\qquad\qquad\qquad\quad \mathsf{else\ let}\ \{xs = g\ (k-1)\ ys\ zs\}\ \mathsf{in}\ [\cdot]\\ \mathsf{in}\ g\ k\ \mathsf{nil}\ \mathsf{nil}\end{array}$$

To get an intuition consider the case when $k = 3$. If we plug $(xs +\!\!+ ys) +\!\!+ zs$ into the hole the evaluation can be thought of as evaluating

$$((((((\mathsf{nil} +\!\!+ ys) +\!\!+ zs) +\!\!+ ys) +\!\!+ zs) +\!\!+ ys) +\!\!+ zs).$$

The computation will require stack but no heap, except for the closures for ys and zs (which is created once and for all) and the short lived closures used to hold the first arguments to *append* and g. If we instead plug $xs +\!\!+ (ys +\!\!+ zs)$ into the hole the computation can be thought of as evaluating

$$((\mathsf{nil} +\!\!+ (ys +\!\!+ zs)) +\!\!+ (ys +\!\!+ zs)) +\!\!+ (ys +\!\!+ zs)$$

which will require 3 heap closures for the 3 occurrences of $(ys +\!\!+ zs)$. When we increase k the difference in heap usage increases and there is no constant which bounds the difference so we have $\Gamma \vdash (xs +\!\!+ ys) +\!\!+ zs \not\underset{\approx}{\triangleright} xs +\!\!+ (ys +\!\!+ zs)$.

Case Study 4: Tail Recursion

This case study is about tail recursion – a transformation very much aimed at improvement in space behaviour. But tail recursive transformations may also improve time complexity and this case study is about such an example. Consider the naive definition of a function that reverses a list:

$$\begin{aligned} \mathit{reverse}\ xs = {} & \mathsf{case}\ xs\ \mathsf{of} \\ & \mathsf{nil} \to \mathsf{nil} \\ & y : ys \to \mathit{reverse}\ ys{+\!+}[y] \end{aligned}$$

The function uses up stack proportional to the length of the list and it also suffers from a quadratic time complexity due to the repeated applications of append. The cure is well-known: transform the function to a tail recursive accumulating parameter definition:

$$\begin{aligned} \mathit{reverse}'\ xs &= \mathit{rev}\ [\,]\ xs \\ \mathit{rev}\ as\ xs &= \mathsf{case}\ xs\ \mathsf{of} \\ & \qquad \mathsf{nil} \to \mathsf{nil} \\ & \qquad y : ys \to \mathit{rev}\ (y : as)\ ys \end{aligned}$$

The tail recursive $\mathit{reverse}'$ has a linear time complexity and the following result confirms our hopes about it's space use.

Proposition 6.7
$\Gamma \vdash \mathit{reverse}\ xs \underset{\approx}{\triangleright} \mathit{reverse}'\ xs$

We will not go into any details about the proof of this proposition but comment on one aspect of the proof. In a proof of contextual equivalence of the two definitions it is helpful to fall back on a result about the associativity of append. Proposition 6.4 provides such a result of weak improvement but it is useless for our proof of Proposition 6.7 because our proof relies on strong improvement. Instead we use the strong improvement in Lemma 6.5. It complicates matters because Lemma 6.5 refers to four different "gadget-versions" $+\!\!+_a \ \ldots \ +\!\!+_d$ of append. This illustrates a general problem: when working with strong improvement we cannot rely on weak improvement results.

Case Study 5: Strict Accumulating Parameters

This case study is about an example where a tail recursion transformation alone does not solve the problem but where we also need a transformation step guided by strictness information.

Consider the naive definition of *sum*.

$$\begin{aligned} sum\ xs = {} & \mathsf{case}\ xs\ \mathsf{of} \\ & \mathsf{nil} \to 0 \\ & y : ys \to y + sum\, ys \end{aligned}$$

The definition suffers from the same problem as the naive definition of *reverse* – it requires stack proportional to the length of the input list. At first it may appear that a plain tail recursion transformation would do the job:

$$\begin{aligned} sum'xs = {} & asum\ 0\ xs \\ asum\ a\ xs = {} & \mathsf{case}\ xs\ \mathsf{of} \\ & \mathsf{nil} \to a \\ & y : ys \to \mathsf{let}\ a' = a + y \\ & \qquad\qquad \mathsf{in}\ asum\ a'\ ys \end{aligned}$$

But sum' still uses stack proportional to the length of its argument: Because of lazy evaluation, the evaluation of $a + y$, in the recursive call of *asum*, is delayed until required. As a result a chain of closures representing the sum builds up in the heap and when the computation is forced it takes up stack proportional to the length of the input list. The next transformation step hinges on the fact that *asum* is strict in the accumulating parameter and forces the accumulator to be computed in each step of the recursion:

$$\begin{aligned} sum''xs = {} & asum\ 0\ xs \\ asum'\ a\ xs = {} & \mathsf{case}\ xs\ \mathsf{of} \\ & \mathsf{nil} \to a \\ & y : ys \to \mathsf{let}\ a' = a + y \\ & \qquad\qquad \mathsf{in}\ \mathsf{seq}\, a'\ (asum'\ a'\ ys) \end{aligned}$$

This is the kind of transformation that a complier with a strictness analyser typically performs. But strictness transformations in general are dangerous from the point of view of space use because they may change evaluation order. Consider, for example the strict function $\lambda y.\lambda x.x+y$. A compiler with strictness analysis might well change the order of the evaluation of the arguments, and from the example in the introduction it should be clear why this is not a space improvement.

Indeed, it happens in this case also: *asum* will traverse the entire spine of its input before evaluating any of its elements, but $asum'$ will evaluate the elements as it traverses the list. The following family of contexts (indexed by k) explores

the difference in evaluation order to show that $\Gamma \vdash sum'\ xs \not\underset{\approx}{\triangleright} sum''\ xs$:

$$\begin{array}{l}\mathsf{let}\ f\ a = \mathsf{nil} \\ \quad ys = fromto\ 1\ k \\ \quad xs = (traverse\ ys) : (f\ ys) \\ \mathsf{in}\ [\cdot]\end{array}$$

where *traverse* is a function that traverses a list and returns 0.

It seems that any transformation which changes the evaluation order of arguments or free variables (or their substructures) can never be a space improvement. At this point it seems that all is lost. However, it is still possible to use strictness transformations as a part of a transformation if it is combined with *another* transformation step which inverts the change made by the strictness phase. This is exactly what happens in this case study! The transformation from *sum* to *sum′* that introduced the accumulating parameter also changes the evaluation order: *sum* evaluates the elements of its input as it traverses the list but *sum′* traverses the entire spine of the list first. As a result this individual transformation step is not space safe either, i.e., $\Gamma \vdash sum\ xs \not\underset{\approx}{\triangleright} sum'\ xs$, which can be shown by a family of contexts similar in spirit to the one above. But taken together the transformations as a whole do not change evaluation order and moreover can be shown to be space safe:

Proposition 6.8
$\Gamma \vdash sum\ xs \underset{\approx}{\triangleright} sum''\ xs$

The proof is along the lines of the previous proofs where we add gadgets to *sum* to obtain:

$$\begin{array}{rl}sum_a\ xs = & \mathsf{let}\ z = \Omega \\ & \mathsf{in\ case}\ xs\ \mathsf{of} \\ & \quad \mathsf{nil} \to {}^{\{z\}}0 \\ & \quad y : ys \to \mathsf{let}\ w = \Omega \\ & \qquad\qquad \mathsf{in}\ {}^{3\curlyvee\{z\}3}(y +^{\{w\}} sum_a \cdot ys)\end{array}$$

The calculation steps in the proof have also been checked by our tool. It is worth noting that we found it very useful in the course of the proof to employ explicit constructs for boxing and unboxing of integers in the language. This allows the proof and the required basic laws to be more fine-grained. The usefulness of these language constructs when performing program transformation is also noted by Peyton Jones and Launchbury [PJL91].

Case Study 6: Tupling

Tupling is the name of a set of program transformation that bring together computations over the same input [Pet77, Chi93]. Tupling transformations can

dramatically reduce the amount of space and time required. Consider for example the naive function to compute the average value of the elements of a list:

$$average\ xs = sum\ xs / length\ xs.$$

The function requires linear space even if *sum* and *length* are space-efficient tail recursive functions. The reason is that (assuming / evaluates from left to right) while *sum* traverses (the lazily produced) input list, the call to *length* holds on to a reference to the start of the list so the entire list will be live. Another example which suffers from the same problem is the naive definition of the function *split* which splits a list of characters into two lists, one containing the first line, and one containing what remains after the first (if any) newline character:

$$split\ xs = (beforeNewline\ xs\ ,\ afterNewline\ xs)$$

where *beforeNewline* and *afterNewline* are defined in the obvious way. A solution to the space problems could be to tuple the computations, i.e., to simultaneously compute the first line and the remainder by a single traversal of the input list. Such a function can be defined as follows.

$$\begin{array}{l} split'\ xs = \mathsf{case}\ xs\ \mathsf{of} \\ \qquad\qquad \mathsf{nil} \rightarrow (\mathsf{nil}, \mathsf{nil}) \\ \qquad\qquad y : ys \rightarrow \mathsf{if}\ y = newline \\ \qquad\qquad\qquad\qquad \mathsf{then}\ (\mathsf{nil}, ys) \\ \qquad\qquad\qquad\qquad \mathsf{else\ let}\ p = split'\ ys \\ \qquad\qquad\qquad\qquad\qquad \mathsf{in}\ (y : (fst\ p), snd\ p) \end{array}$$

Note that *split'*, in contrast to *split*, is strict. However, this definition doesn't solve the problem. The reason is the use of the projections *fst p* and *snd p*. Due to lazy evaluation, the projections are not evaluated until needed and therefore hold on to the reference to p, which in turn holds on to both the results of the recursive call. As a result, we have combined not only the computations but also the lifetimes of the two results.

Intriguingly, this problem appears to be linked to the intensional expressiveness of the language. Hughes has argued that it is impossible to define *split* in a space efficient way using a particular lazy evaluator [Hug83]. He proposed a solution involving combinators for explicit parallelism and synchronisation. With these language primitives the original definition of *split* can be made efficient by having just the right degree of parallelism. Another proposal, due to Wadler [Wad87], is to solve the problem by extending the garbage collector. Whenever the garbage collector encounters a term of the form *fst p* where p is bound to an evaluated pair, it may perform the reduction of the projection. A more recent proposal is due to Sparud [Spa93]. He proposes to treat *pattern bindings* in let expressions specially. A pattern binding in a let expression takes the form

$$\mathsf{let}\ \{c\,\vec{x} = M\}\ \mathsf{in}\ N.$$

Prior to Sparud's proposal, these kind of bindings were thought of as mere syntactic sugar and a compiler (e.g. [Aug87]) would typically translate it into the following

$$\mathsf{let}\ \{p = M, x_1 = \Pi_{c_1}\, p, \ldots, x_n = \Pi_{c_n}\, p\}\ \mathsf{in}\ N$$

which reintroduces the "dangerous" projections.

Sparud's proposal was to have pattern bindings as a first class construct which the evaluator treats in a space efficient manner. We have adopted Sparud's proposal because we think it is the most natural and because it leads to a reasonably well behaved space theory. Implementing Wadler's proposal in our model of garbage collection would destroy many of the nice properties of our theory. For example, beta-expansion would no longer be space safe, because it may result in the elimination of a "garbage collector redex".

We have formalised Sparuds proposal as an extension to our language. The extension can be found in Appendix B. With pattern bindings at hand we can rewrite *split'* as follows.

$$\begin{array}{l} \mathit{split''}\ \mathit{xs} = \mathsf{case}\ \mathit{xs}\ \mathsf{of} \\ \qquad\qquad \mathsf{nil} \to (\mathsf{nil}, \mathsf{nil}) \\ \qquad\qquad \mathit{y} : \mathit{ys} \to \mathsf{if}\ \mathit{y} = \mathit{newline} \\ \qquad\qquad\qquad\qquad \mathsf{then}\ (\mathsf{nil}, \mathit{ys}) \\ \qquad\qquad\qquad\qquad \mathsf{else}\,\mathsf{let}\ (\mathit{ps}, \mathit{qs}) = \mathit{split''}\ \mathit{ys} \\ \qquad\qquad\qquad\qquad\quad \mathsf{in}\ (\mathit{y} : \mathit{ps}, \mathit{qs}) \end{array}$$

So, what is the relation between the different versions of *split*? Let us start with the relation between *split'* and *split''* where we have that $\Gamma \vdash \mathit{split'}\ \mathit{xs} \gtrapprox \mathit{split''}\ \mathit{xs}$. It follows directly from the following lemma.

Lemma 6.9

$$\begin{array}{l} \mathsf{let}\ \{{}_2p = M\}\ \mathsf{in}\ \mathbb{C}[\mathit{fst}\ p][\mathit{snd}\ p] \\ \qquad\qquad \gtrapprox\ \mathsf{let}\ \{(x, y) = M\}\ \mathsf{in}\ \mathbb{C}[x][y] \quad \text{if } p \notin \mathsf{FV}(M, \mathbb{C}) \end{array}$$

The lemma says that it always space safe to use pattern bindings instead of projections. So what about *split* and *split''*? Convinced that

$$\Gamma \vdash \mathsf{let}\ \{(x, y) = \mathit{split}\ \mathit{xs}\}\ \mathsf{in}\ M \gtrapprox \mathsf{let}\ \{(x, y) = \mathit{split''}\ \mathit{xs}\}\ \mathsf{in}\ M$$

we spent considerable effort trying to prove it only to realise that it is not the case. The family of contexts that distinguishes between *split xs* and *split'' xs* is somewhat involved so we found it better to present the intuition about why *split'' xs* in some contexts may use more space than *split xs*.

Consider a context where the second component of the pair is used before the first, i.e., a program which processes the second line of its input before the first.

In that case the tupling has the effect that the spine of the list representing the first line of input is constructed *before* it is needed (in our definition of *split''* this allocation is hidden in the syntactic sugar). This in itself does not lead to a non constant factor worsening if the spine of the input list may be garbage collected. But what if it can't? Consider a program which processes its second line of input repeatedly and selects the line from the input by repeatedly applying *split''* to the input. Suppose also that it keeps references to the different copies of the first line that is constructed. Such a context, however unlikely in practice, would show that $\Gamma \vdash \mathit{split}\ \mathit{xs} \not\gtrapprox \mathit{split''}\ \mathit{xs}$.

This have lead us to the general observation that tupling of computations which need to allocate space in order to produce its output are unlikely to be space improvements, although we have not been able to make this statement more precise.

Another observation, at this point maybe not surprising, is that tupling transformations which change the order in which inputs (or the substructures thereof) are traversed are unlikely to be space improvements. The tupling of the sum and the length of a list is an example of this. In a context where the length of the list is needed before the sum, the untupled definition would traverse the spine of the list before any of the elements, but the tupled definition would force the computation of the elements as it traverses the list. These two observations have made us rather pessimistic about showing that tupled function improve on their untupled counterparts. However, in contexts which are guaranteed to require the result of the tupled computation in a specific order the situation may be different. For example, we believe that for *average'* defined using a tupled computation of the sum and the length we would have $\Gamma \vdash \mathit{average}\ \mathit{xs} \gtrapprox \mathit{average'}\ \mathit{xs}$) because the functions (due to the evaluation order of /) requires the sum before the length.

7 Proof of the Context Lemma

In this section we prove the context lemma of strong improvement – our key technical vehicle for establishing laws of improvement. We will also introduce some technical machinery which will be useful also in the proofs of some of the laws of strong improvement in Section 8.

7.1 Generalised Contexts

Before we can proceed with the proof we need to generalise the notion of contexts and extend our semantics to computation with contexts. We use a second-order syntax for contexts which is due to Pitts [Pit94]. A detailed account of this approach can be found in [San98]. Generalised contexts may have several holes each of which may occur zero of more times. To distinguish between the different holes we use hole variables ranged over by ξ. The holes also take a different form

$$
\begin{array}{lcl}
\xi\,\vec{x}[\Lambda\vec{y}.M/\xi] & \stackrel{\text{def}}{=} & M[\vec{x}/\vec{y}] \\
x[\Xi/\xi] & \stackrel{\text{def}}{=} & x \\
(\lambda x.\mathbb{M})[\Xi/\xi] & \stackrel{\text{def}}{=} & \lambda x.\mathbb{M}[\Xi/\xi] \qquad \text{if } x \notin \mathsf{FV}(\Xi) \\
\mathbb{M}\,x[\Xi/\xi] & \stackrel{\text{def}}{=} & \mathbb{M}[\Xi/\xi]\,x \\
n[\Xi/\xi] & \stackrel{\text{def}}{=} & n \\
(\mathbb{M}+\mathbb{N})[\Xi/\xi] & \stackrel{\text{def}}{=} & \mathbb{M}[\Xi/\xi]+\mathbb{N}[\Xi/\xi] \\
(\mathsf{add}_n\mathbb{M})[\Xi/\xi] & \stackrel{\text{def}}{=} & \mathsf{add}_n\mathbb{M}[\Xi/\xi] \\
(\mathsf{iszero}\,\mathbb{M})[\Xi/\xi] & \stackrel{\text{def}}{=} & \mathsf{iszero}\,\mathbb{M}[\Xi/\xi] \\
(\mathsf{seq}\,\mathbb{M}\,\mathbb{N})[\Xi/\xi] & \stackrel{\text{def}}{=} & \mathsf{seq}\,\mathbb{M}[\Xi/\xi]\,\mathbb{N}[\Xi/\xi] \\
(\mathsf{let}\ \{\vec{x}=\vec{\mathbb{M}}\}\ \mathsf{in}\ \mathbb{N})[\Xi/\xi] & \stackrel{\text{def}}{=} & \mathsf{let}\ \{\vec{x}=\vec{\mathbb{M}}[\Xi/\xi]\}\ \mathsf{in}\ \mathbb{N}[\Xi/\xi] \quad \text{if } \{\vec{x}\}\cap\mathsf{FV}(\Xi)=\emptyset \\
(\mathsf{case}\ \mathbb{M}\ \mathsf{of}\ \{c_i\,\vec{x}_i\to\mathbb{N}_i\})[\Xi/\xi] & \stackrel{\text{def}}{=} & \mathsf{case}\ \mathbb{M}[\Xi/\xi]\ \mathsf{of}\ \{c_i\,\vec{x}_i\to\mathbb{N}_i[\Xi/\xi]\} \quad \text{if } \{\vec{x}_i\}\cap\mathsf{FV}(\Xi)=\emptyset
\end{array}
$$

Figure 9: Hole filling for generalised contexts.

in generalised contexts: instead of plain holes $[\cdot]$, in generalised contexts each occurrence of a hole ξ is applied to a vector of variables: $\xi\,\vec{x}$. The grammar of generalised context is

$$
\begin{array}{rcl}
\mathbb{L},\mathbb{M},\mathbb{N} & ::= & \xi\,\vec{x} \\
& \mid & x \mid \lambda x.\mathbb{M} \mid {}^{w}(\mathbb{M}\,x) \mid c\,\vec{x} \mid {}^{w}(\mathsf{seq}\,\mathbb{M}\,\mathbb{N}) \\
& \mid & n \mid {}^{w_0}(\mathbb{M} +^{w_1} \mathbb{N}) \mid {}^{w}(\mathsf{add}_n\mathbb{M}) \mid {}^{w}(\mathsf{iszero}\,\mathbb{M}) \\
& \mid & \mathsf{let}\ \{{}^{v_i}_{w_i}x_i = \mathbb{M}_i\}_{i\in I}\ \mathsf{in}\ \mathbb{N} \\
& \mid & {}^{w}(\mathsf{case}\ \mathbb{M}\ \mathsf{of}\ \{c_i\,\vec{x}_i\to\mathbb{N}_i\}).
\end{array}
$$

Each hole variable ξ has a fixed arity n and at each occurrence ξ must be applied to a vector of length n. We will identify generalised contexts up to the renaming of bound variables. For conventional contexts α-conversion doesn't make any sense which is the primary reason for why they are not appropriate for the technical development in this paper.

A hole can be filled with a *plug* which is an abstraction of the form $\Lambda\vec{y}.M$. We will let Ξ range over plugs and the arity of a plug $\Lambda\vec{y}.M$ is the length of $\vec{y}$. When plugging $\Lambda\vec{y}.M$ into a hole $\xi\,\vec{x}$ with the same arity the result is the term $M[\vec{x}/\vec{y}]$. We will write $\mathbb{M}[\Xi/\xi]$ for the operation of filling all the occurrences of the hole ξ in $\mathbb{M}$ with the plug Ξ. The definition is analogous to the definition of ordinary non-capturing substitution and is in Figure 9. Analogously to term contexts, we have heap contexts $\mathbb{\Gamma}$ and stack contexts $\mathbb{S}$ which are heaps and stack with holes. A stack context consists of update markers and reduction contexts with holes – reduction context contexts if you like. They are defined

$$\langle\, \mathbb{\Gamma}\{^{v}_{w} x = \mathbb{M}\},\ x,\ \mathbb{S}\,\rangle \to \langle\, \mathbb{\Gamma},\ \mathbb{M},\ \#^{v}_{w} x : \mathbb{S}\,\rangle \qquad (\textit{Lookup})$$

$$\langle\, \mathbb{\Gamma},\ \mathbb{V},\ \#^{v}_{w} x : \mathbb{S}\,\rangle \to \langle\, \mathbb{\Gamma}\{^{v}_{w} x = \mathbb{V}\},\ \mathbb{V},\ \mathbb{S}\,\rangle \qquad (\textit{Update})$$

$$\langle\, \mathbb{\Gamma},\ \mathsf{let}\ \mathbb{\Gamma}'\ \mathsf{in}\ \mathbb{N},\ \mathbb{S}\,\rangle \to \langle\, \mathbb{\Gamma}\mathbb{\Gamma}',\ \mathbb{N},\ \mathbb{S}\,\rangle \qquad (\textit{Letrec})$$

$$\langle\, \mathbb{\Gamma},\ {}^{w}\mathbb{R}[\mathbb{M}],\ \mathbb{S}\,\rangle \to \langle\, \mathbb{\Gamma},\ \mathbb{M},\ {}^{w}\mathbb{R} : \mathbb{S}\,\rangle \qquad (\textit{Push})$$

$$\langle\, \mathbb{\Gamma},\ \mathbb{V},\ {}^{w}\mathbb{R} : \mathbb{S}\,\rangle \to \langle\, \mathbb{\Gamma},\ \mathbb{M},\ \mathbb{S}\,\rangle \quad \text{if } \mathbb{R}[\mathbb{V}] \rightsquigarrow \mathbb{M} \qquad (\textit{Reduce})$$

$$\begin{aligned}
(\lambda x.\mathbb{M})\, y &\rightsquigarrow \mathbb{M}[{}^{y}/_{x}] \\
\mathsf{case}\ c_j\, \vec{y}\ \mathsf{of}\ \{c_i\, \vec{x}_i \to \mathbb{M}_i\} &\rightsquigarrow \mathbb{M}_j[{}^{\vec{y}}/_{\vec{x}_j}] \\
m +^{w} \mathbb{N} &\rightsquigarrow {}^{w}\mathsf{add}_m \mathbb{N} \\
\mathsf{add}_m n &\rightsquigarrow \ulcorner m + n \urcorner \\
\mathsf{iszero}\, m &\rightsquigarrow \begin{cases} \mathsf{true} & \text{if } m = 0 \\ \mathsf{false} & \text{otherwise} \end{cases} \\
\mathsf{seq}\, \mathbb{V}\, \mathbb{M} &\rightsquigarrow \mathbb{M}
\end{aligned}$$

Figure 10: Abstract machine semantics for contexts

thus.

$$\begin{aligned}
\mathbb{R} ::= {} & [\cdot]\, x \mid \mathsf{case}\ [\cdot]\ \mathsf{of}\ \{c_i\, \vec{x}_i \to \mathbb{N}_i\} \mid \mathsf{seq}\, [\cdot]\, \mathbb{M} \mid \\
& [\cdot] + \mathbb{M} \mid \mathsf{add}_n [\cdot] \mid \mathsf{iszero}\, [\cdot]
\end{aligned}$$

We will also have abstract machine configurations with holes and let $\mathbb{\Sigma}$ and $\mathbb{\Phi}$ range over such configuration contexts.

7.2 Computing with contexts

In Figure 10 we have lifted the definition of the abstract machine to configuration contexts. A key property is that hole filling commutes with transitions:

Lemma 7.1
If $\mathbb{\Sigma} \to \mathbb{\Phi}$ then $\mathbb{\Sigma}[{}^{\Xi}/_{\xi}] \to \mathbb{\Phi}[{}^{\Xi}/_{\xi}]$.

Proof. By inspection of the rules. □

We have arrived to the definition of the abstract machine simply by replacing all occurrences of Γ with $\mathbb{\Gamma}$, M with $\mathbb{M}$ and so on, in the original definition. In [San98] Sands argues that for a certain form of syntax oriented definitions, this will always result in relations which commute with hole-filling, by virtue of the representation of contexts.

However, our accessibility-based definition of garbage collection does not fit that format. Recall that garbage collection is the removal of bindings and update markers such that the configuration remains closed. Thus the definition relies on the notion of free variables. How can we lift the definition of free variables to contexts? Our definition, which we think is the only reasonable one, have the clause

$$\mathsf{FV}(\xi\,\vec{x}) = \{\vec{x}\}.$$

This definition does not commute with hole filling. For example:

$$\mathsf{FV}(\xi\,x\,y) = \{x,\ y\}$$

but

$$\mathsf{FV}(\xi\,x\,y[\Lambda x'\,y'.y' + z/\xi]) = \mathsf{FV}(y + z) = \{y,\ z\}.$$

As the example illustrates the definition fails to commute for two reason: firstly the plug may contain free variables (z in our example) and secondly the plug may ignore some of its arguments (x in our example). To make hole-filling commute with garbage collection we need to restrict which plugs can be plugged in. If we restrict the plug to have no free variables, then the free variables of a context is always a superset of the free variables after filling the hole:

Lemma 7.2
For every closed plug Ξ, $\mathsf{FV}(\mathbb{M}) \supseteq \mathsf{FV}(\mathbb{M}[\Xi/\xi])$.

Proof. By induction over the structure of $\mathbb{M}$. □

This restriction is enough for hole-filling to commute with garbage collection:

Lemma 7.3
For a closed plug Ξ, if $\mathbb{\Sigma} \gg \Phi$ then $\mathbb{\Sigma}[\Xi/\xi] \gg \Phi[\Xi/\xi]$.

Proof. Immediate by lemma 7.3. □

It is a simple consequence that, for closed plugs, hole-filling commutes with computation:

Lemma 7.4
For every closed plug Ξ,

- if $\mathbb{\Sigma} \rightarrow^n_{(h,s)} \Phi$ then $\mathbb{\Sigma}[\Xi/\xi] \rightarrow^n_{(h,s)} \Phi[\Xi/\xi]$,
- if $\mathbb{\Sigma}\Downarrow_{(h,s)}$ then $\mathbb{\Sigma}[\Xi/\xi]\Downarrow_{(h,s)}$.

Proof. By induction over the length of the computation. □

The restriction to closed plugs is not enough to ensure that the free variables of a context coincides with the free variables after filling the hole. With the additional restriction that a plug uses all its arguments, we say that such a plug is linear, we get the desired property:

Lemma 7.5
For a closed linear plug Ξ, $\mathsf{FV}(\mathbb{M}) = \mathsf{FV}(\mathbb{M}[\Xi/\xi])$.

PROOF. By induction over the structure of $\mathbb{M}$. □

The additional restriction is necessary for the Uniform Computation lemma.

Lemma 7.6 (Uniform Computation)
For every closed linear plug Ξ, if $\mathbb{\Sigma}[\Xi/\xi]\Downarrow^n_{(h,s)}$ then

- either $\mathbb{\Sigma}\Downarrow^n_{(h,s)}$,
- or there exists m, $\mathbb{\Gamma}$, $\vec{x}$ and $\mathbb{S}$ such that
  - $\mathbb{\Sigma} \twoheadrightarrow^m_{(h,s)} \langle\, \mathbb{\Gamma},\ \xi\,\vec{x},\ \mathbb{S}\,\rangle$ and
  - $\langle\, \mathbb{\Gamma},\ \xi\,\vec{x},\ \mathbb{S}\,\rangle[\Xi/\xi]\Downarrow^{n-m}_{(h,s)}$.

PROOF. By induction over the length of the computation. □

The significance of the lemma is that either a computation doesn't depend on a subterm or if it does the computation can run until the subterm is in the evaluation position of the configuration. The restriction to linear plugs is necessary because otherwise the configuration context may hold on to additional references and thus possibly require more space.

7.3 An Auxiliary Lemma

On the way to the proof of the context lemma we will show the following auxiliary lemma which contains most of the technical difficulties.

Lemma 7.7
For every closed linear plug $\Lambda\vec{x}.M$, and closed plug $\Lambda\vec{x}.N$, if for all Γ, S, σ,

$$\langle\, \Gamma,\ M\sigma,\ S\,\rangle\Downarrow_{(h,s)} \implies \langle\, \Gamma,\ N\sigma,\ S\,\rangle\Downarrow_{(h,s)}$$

then for all $\mathbb{\Gamma}$, $\mathbb{S}$ and σ such that $\mathbb{\Gamma}$ and $\mathbb{S}$ may contain ξ but no other hole,

$$\langle\, \mathbb{\Gamma},\ M\sigma,\ \mathbb{S}\,\rangle[\Lambda\vec{x}.M/\xi]\Downarrow_{(h,s)} \implies \langle\, \mathbb{\Gamma},\ N\sigma,\ \mathbb{S}\,\rangle[\Lambda\vec{x}.N/\xi]\Downarrow_{(h,s)}$$

PROOF. Assume the premise. We will show by induction over n that, for all $\mathbb{\Gamma}$, $\mathbb{S}$ and σ such that $\mathbb{\Gamma}$ and $\mathbb{S}$ contains no hole but ξ,

$$\langle\, \mathbb{\Gamma},\ M\sigma,\ \mathbb{S}\,\rangle[\Lambda\vec{x}.M/\xi]\Downarrow^n_{(h,s)} \implies \langle\, \mathbb{\Gamma},\ N\sigma,\ \mathbb{S}\,\rangle[\Lambda\vec{x}.N/\xi]\Downarrow_{(h,s)}.$$

Thus assume

$$\langle\, \mathbb{\Gamma},\ M\sigma,\ \mathbb{S}\,\rangle[\Lambda\vec{x}.M/\xi]\Downarrow^n_{(h,s)}.$$

By the Uniform Computation Lemma we know that either

$$\langle\, \mathbb{\Gamma},\ M\sigma,\ \mathbb{S}\,\rangle\Downarrow^n_{(h,s)},$$

or there exists m, $\mathbb{\Delta}$, $\vec{y}$ and $\mathbb{T}$ such that

$$\langle\, \mathbb{\Gamma},\ M\sigma,\ \mathbb{S}\,\rangle \twoheadrightarrow^m_{(h,s)} \langle\, \mathbb{\Delta},\ \xi\,\vec{y},\ \mathbb{T}\,\rangle$$

and

$$\langle\, \mathbb{\Delta},\ \xi\,\vec{y},\ \mathbb{T}\,\rangle[\Lambda\vec{x}.M/\xi]\Downarrow^{n-m}_{(h,s)}.$$

We will first show that in either case

$$\langle\, \mathbb{\Gamma},\ M\sigma,\ \mathbb{S}\,\rangle[\Lambda\vec{x}.N/\xi]\Downarrow_{(h,s)}.$$

In the first case it follows immediately by filling the hole with $\Lambda\vec{x}.N$. Consider the second case. Note that

$$\langle\, \mathbb{\Delta},\ \xi\,\vec{y},\ \mathbb{T}\,\rangle[\Lambda\vec{x}.M/\xi] \equiv \langle\, \mathbb{\Delta},\ M[\vec{y}/\vec{x}],\ \mathbb{T}\,\rangle[\Lambda\vec{x}.M/\xi]$$

so we have that

$$\langle\, \mathbb{\Delta},\ M[\vec{y}/\vec{x}],\ \mathbb{T}\,\rangle[\Lambda\vec{x}.M/\xi]\Downarrow^{n-m}_{(h,s)}.$$

We also have that $m > 0$ since $\langle\, \mathbb{\Gamma},\ M\sigma,\ \mathbb{S}\,\rangle \not\equiv \langle\, \mathbb{\Delta},\ \xi\,\vec{y},\ \mathbb{T}\,\rangle$ so we can apply the induction hypothesis which yields

$$\langle\, \mathbb{\Delta},\ N[\vec{y}/\vec{x}],\ \mathbb{T}\,\rangle[\Lambda\vec{x}.N/\xi]\Downarrow_{(h,s)}.$$

From $\langle\, \mathbb{\Gamma},\ M\sigma,\ \mathbb{S}\,\rangle \twoheadrightarrow^m_{(h,s)} \langle\, \mathbb{\Delta},\ \xi\,\vec{y},\ \mathbb{T}\,\rangle$ it follows by hole filling that

$$\langle\, \mathbb{\Gamma},\ M\sigma,\ \mathbb{S}\,\rangle[\Lambda\vec{x}.N/\xi] \twoheadrightarrow^m_{(h,s)} \langle\, \mathbb{\Delta},\ \xi\,\vec{y},\ \mathbb{T}\,\rangle[\Lambda\vec{x}.N/\xi]$$

so we can conclude that also in the second case we have

$$\langle\, \mathbb{\Gamma},\ M\sigma,\ \mathbb{S}\,\rangle[\Lambda\vec{x}.N/\xi]\Downarrow_{(h,s)}.$$

Note that

$$\langle\, \mathbb{\Gamma},\ M\sigma,\ \mathbb{S}\,\rangle[\Lambda\vec{x}.N/\xi] \equiv \langle\, \mathbb{\Gamma}[\Lambda\vec{x}.N/\xi],\ M\sigma,\ \mathbb{S}[\Lambda\vec{x}.N/\xi]\,\rangle$$

and since ξ is the only hole in $\mathbb{\Gamma}$ and $\mathbb{S}$ we can apply the main assumption which gives

$$\langle\, \mathbb{\Gamma}[\Lambda\vec{x}.N/\xi],\ N\sigma,\ \mathbb{S}[\Lambda\vec{x}.N/\xi]\,\rangle\Downarrow_{(h,s)}.$$

We conclude the proof by noting that

$$\langle\, \mathbb{\Gamma}[\Lambda\vec{x}.N/\xi],\ N\sigma,\ \mathbb{S}[\Lambda\vec{x}.N/\xi]\,\rangle \equiv \langle\, \mathbb{\Gamma},\ N\sigma,\ \mathbb{S}\,\rangle[\Lambda\vec{x}.N/\xi]$$

so we have shown what is required. □

7.4 The context lemma

Before we proceed with the proof we restate the context lemma.

Lemma 7.8 (context lemma)
For all M and N such that $\mathsf{FV}(M) \supseteq \mathsf{FV}(N)$, if for all Γ, S and σ,

$$\langle\, \Gamma,\ M\sigma,\ S\,\rangle\Downarrow_{(h,s)} \implies \langle\, \Gamma,\ N\sigma,\ S\,\rangle\Downarrow_{(h,s)}$$

then $M \gtrsim N$.

Proof. Assume the premise. Let $\vec{x}$ be a vector with the free variables of M. Then $\Lambda\vec{x}.M$ is a closed linear plug. Also, since $\mathsf{FV}(M) \supseteq \mathsf{FV}(N)$, $\Lambda\vec{x}.N$ is a closed plug. Now given arbitrary conventional context $\mathbb{C}$ and σ such that $\mathbb{C}[M\sigma]$ and $\mathbb{C}[N\sigma]$ are closed, and

$$\mathbb{C}[M\sigma]\Downarrow^n_{(h,s)}.$$

We will start by representing the filling of the hole in a conventional context by the filling of a hole in a generalised context. Let $[\vec{y}/\vec{x}]$ be the restriction of σ to the domain $\{\vec{x}\}$ so that $M\sigma \equiv M[\vec{y}/\vec{x}]$ and $N\sigma \equiv N[\vec{y}/\vec{x}]$. Also, let $\mathbb{M}$ be the generalised context which is the result of filling all occurrences of the hole $[\cdot]$ in the conventional context $\mathbb{C}$ with $\xi\,\vec{y}$ (The details of this operation is given in [San98]). It is easy to show that

$$\mathbb{C}[M\sigma] \equiv \mathbb{M}[\Lambda\vec{x}.M/\xi]$$

and

$$\mathbb{C}[N\sigma] \equiv \mathbb{M}[\Lambda\vec{x}.N/\xi].$$

So we have represented the operation of filling the hole in a conventional context by a generalised context. It is worth noting that this construction differs from the one in [San98] which doesn't result in linear nor closed plugs. With this construction we can proceed with the main argument. Recall that

$$\mathbb{C}[M\sigma]\Downarrow^n_{(h,s)},$$

so

$$\langle\, \emptyset,\ \mathbb{M},\ \epsilon\, \rangle[^{\Lambda\vec{x}.M}/_{\xi}]\Downarrow^{n}_{(h,s)}.$$

We know by the Uniform Computation Lemma that, either

$$\langle\, \emptyset,\ \mathbb{M},\ \epsilon\, \rangle\Downarrow^{n}_{(h,s)}$$

or there exists m, $\mathbb{\Gamma}$, $\vec{z}$, $\mathbb{S}$ such that

$$\langle\, \emptyset,\ \mathbb{M},\ \epsilon\, \rangle \twoheadrightarrow^{m}_{(h,s)} \langle\, \mathbb{\Gamma},\ \xi\, \vec{z},\ \mathbb{S}\, \rangle$$

and

$$\langle\, \mathbb{\Gamma},\ \xi\, \vec{z},\ \mathbb{S}\, \rangle[^{\Lambda\vec{x}.M}/_{\xi}]\Downarrow^{n-m}_{(h,s)}.$$

In the first case the required result follows by filling the hole with $\Lambda\vec{x}.N$. Consider the second case. Note that

$$\langle\, \mathbb{\Gamma},\ \xi\, \vec{z},\ \mathbb{S}\, \rangle[^{\Lambda\vec{x}.M}/_{\xi}] \equiv \langle\, \mathbb{\Gamma},\ M[^{\vec{z}}/_{\vec{x}}],\ \mathbb{S}\, \rangle[^{\Lambda\vec{x}.M}/_{\xi}]$$

so it follows by Lemma 7.7 that

$$\langle\, \mathbb{\Gamma},\ N[^{\vec{z}}/_{\vec{x}}],\ \mathbb{S}\, \rangle[^{\Lambda\vec{x}.N}/_{\xi}]\Downarrow_{(h,s)}$$

By filling the hole in $\langle\, \emptyset,\ \mathbb{M},\ \epsilon\, \rangle \twoheadrightarrow^{m}_{(h,s)} \langle\, \mathbb{\Gamma},\ \xi\, \vec{z},\ \mathbb{S}\, \rangle$ with $\Lambda\vec{x}.N$ we know that

$$\langle\, \emptyset,\ \mathbb{M},\ \epsilon\, \rangle[^{\Lambda\vec{x}.N}/_{\xi}] \twoheadrightarrow^{m}_{(h,s)} \langle\, \mathbb{\Gamma},\ \xi\, \vec{z},\ \mathbb{S}\, \rangle[^{\Lambda\vec{x}.N}/_{\xi}] \equiv \langle\, \mathbb{\Gamma},\ N[^{\vec{z}}/_{\vec{x}}],\ \mathbb{S}\, \rangle[^{\Lambda\vec{x}.N}/_{\xi}]\Downarrow_{(h,s)}$$

which concludes the proof of the context lemma. □

8 Proofs of Selected Laws

In this section we present detailed proofs of some laws of strong improvement. We have chosen to present the proofs of the laws *reduction*, *let-R*, *R-case*, *let-alts* and *let-let*. The first two represent the majority of laws which can be proved with the help of the context lemma in a rather straightforward manner. The other three present additional difficulties and require additional technical machinery.

8.1 Proof of *reduction*

In this section we present the proof of the most fundamental law of strong improvement:

$$^{w}R[V] \mathrel{\lessapprox\atop\gtrsim} {}^{w \curlyvee X}N \qquad \text{if } R[V] \rightsquigarrow N \text{ and } \mathsf{FV}(R[V]) = \mathsf{FV}(^{X}N) \qquad (\textit{reduction})$$

The context lemma makes the law very easy to establish. It is easy because when the two terms are placed in the evaluation position of a configuration the respective configurations reduces to the same configuration up to garbage in a few steps.

The right-way improvement We start with the right-way improvement, i.e.,

$$ {}^{w}R[V] \gtrsim {}^{w \curlyvee X}N \qquad \text{if } R[V] \rightsquigarrow N \text{ and } \mathsf{FV}(R[V]) = \mathsf{FV}({}^{X}N) \ , $$

which is the easiest.

Proof. Assume the side conditions i.e., that

$$ R[V] \rightsquigarrow N $$

and

$$ \mathsf{FV}(R[V]) = \mathsf{FV}({}^{X}N). $$

By the context lemma it is enough to show that for all Γ_0, S_0 and σ,

$$ \langle\, \Gamma_0,\ {}^{w}R[V]\sigma,\ S_0 \,\rangle\Downarrow_{(h,s)} \implies \langle\, \Gamma_0,\ ({}^{w \curlyvee X}N)\sigma,\ S_0 \,\rangle\Downarrow_{(h,s)}. $$

Thus assume

$$ \langle\, \Gamma_0,\ {}^{w}R[V]\sigma,\ S_0 \,\rangle\Downarrow_{(h,s)} $$

i.e., that

$$ \begin{array}{ll} & \langle\, \Gamma_0,\ {}^{w}R[V]\sigma,\ S_0 \,\rangle \\ \to_{(h,s)} & \langle\, \Gamma_0,\ V\sigma,\ {}^{w}R\sigma : S_0 \,\rangle \\ \gtrsim & \langle\, \Gamma_1,\ V\sigma,\ {}^{w}R\sigma : S_1 \,\rangle \\ \to_{(h,s)} & \langle\, \Gamma_1,\ N\sigma,\ S_1 \,\rangle \\ \gtrsim & \langle\, \Gamma_2,\ N\sigma,\ S_2 \,\rangle\Downarrow_{(h,s)}. \end{array} $$

It follows immediately by the definition of transitions and garbage collection, using $\mathsf{FV}(R[V]) = \mathsf{FV}({}^{X}N)$, that

$$ \begin{array}{lll} & \langle\, \Gamma_0,\ ({}^{w \curlyvee X}N)\sigma,\ S_0 \,\rangle & \\ \equiv & \langle\, \Gamma_0,\ ({}^{w}\mathsf{case}\ \mathsf{true}\ \mathsf{of}\ \{\mathsf{true} \to {}^{X}N\})\sigma,\ S_0 \,\rangle & \\ \to & \langle\, \Gamma_0,\ \mathsf{true},\ {}^{w}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{\mathsf{true} \to ({}^{X}N)\sigma\} : S_0 \,\rangle & \\ \gtrsim & \langle\, \Gamma_1,\ \mathsf{true},\ {}^{w}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{\mathsf{true} \to ({}^{X}N)\sigma\} : S_1 \,\rangle & \\ \to & \langle\, \Gamma_1,\ ({}^{X}N)\sigma,\ S_1 \,\rangle & \\ \gtrsim & \langle\, \Gamma_1,\ ({}^{X}N)\sigma,\ S_1 \,\rangle & \\ \equiv & \langle\, \Gamma_1,\ (\mathsf{let}\ \{\vec{y} = \vec{x}\}\ \mathsf{in}\ N)\sigma,\ S_1 \,\rangle & \text{where } X = \{\vec{x}\} \text{ and } \vec{y} \text{ fresh} \\ \to & \langle\, \Gamma_1\{\vec{y} = \vec{x}\sigma\},\ N\sigma,\ S_1 \,\rangle & \\ \gtrsim & \langle\, \Gamma_2,\ N\sigma,\ S_2 \,\rangle\Downarrow_{(h,s)}. & \end{array} $$

Note that ${}^{X}N$ is defined in terms of a let expression which allocates a set of bindings which we have used above. The bindings can be collected immediately so they do not present any difficulty in this case. We will see that when we prove the left-way improvement they lead to a small complication. It remains

to show that the configurations in the transition sequence uses at most (h, s) space. For the first configuration we have

$$|\langle\, \Gamma_0,\ ({}^{w\curlyvee X}N)\sigma,\ S_0\,\rangle| = |\langle\, \Gamma_0,\ {}^{w}R[V]\sigma,\ S_0\,\rangle| \leq (h, s),$$

for the second

$$|\langle\, \Gamma_1,\ {}^{w}\mathsf{true},\ {}^{w}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{\mathsf{true} \to ({}^{X}N)\sigma\} : S_1\,\rangle| = |\langle\, \Gamma_1,\ V\sigma,\ {}^{w}R\sigma : S_1\,\rangle| \leq (h, s),$$

and for the third

$$|\langle\, \Gamma_1,\ ({}^{X}N)\sigma,\ S_1\,\rangle| < |\langle\, \Gamma_1,\ V\sigma,\ {}^{w}R\sigma : S_1\,\rangle| \leq (h, s).$$

Thus

$$\langle\, \Gamma_0,\ ({}^{w\curlyvee X}N)\sigma,\ S_0\,\rangle\Downarrow_{(h,s)}$$

as required. □

The left-way improvement The left-way improvement

$${}^{w}R[V] \lesssim {}^{w\curlyvee X}N \qquad \text{if } R[V] \rightsquigarrow N \text{ and } \mathsf{FV}(R[V]) = \mathsf{FV}({}^{X}N)$$

presents a small complication because of the way we have defined ${}^{\{\vec{x}\}}M$:

$${}^{\{\vec{x}\}}M \stackrel{\text{def}}{=} \mathsf{let}\ \{\vec{y} = \vec{x}\}\ \mathsf{in}\ M \qquad \text{where } \vec{y} \text{ are fresh.}$$

The bindings that are allocated by the let can be garbage collected immediately and if so they do not take up space. But garbage collection is non-deterministic so they may be kept. So if we compute with $\langle\, \Gamma,\ {}^{w\curlyvee X}N,\ S\,\rangle$ there may be no way for $\langle\, \Gamma,\ {}^{w}R[V],\ S\,\rangle$ to reduce to an identical configuration because there is no way to introduce the additional garbage. To deal with this we need the following lemma.

Lemma 8.1
If $\Sigma\Downarrow_{(h,s)}$ and $\Sigma \gg \Sigma'$ then $\Sigma'\Downarrow_{(h,s)}$.

Proof. The lemma is proved by induction over the length of the computation. In each step we need to show that the missing garbage does not effect the transition. A subtlety is that garbage in the stack, i.e., dead update markers, can effect transitions: the computation with Σ may perform some updates which Σ' can't. But these updates always yields bindings which are garbage so they can't effect the outcome of the computation. □

With this lemma we are ready to show the left-way improvement.

PROOF. Assume the side conditions i.e., that

$$R[V] \rightsquigarrow N$$

and

$$\mathsf{FV}(R[V]) = \mathsf{FV}({}^{X}N).$$

By the context lemma it is enough to show that for all Γ_0, S_0 and σ,

$$\langle\, \Gamma_0,\ ({}^{w \curlyvee X}N)\sigma,\ S_0 \,\rangle\Downarrow_{(h,s)} \implies \langle\, \Gamma_0,\ {}^{w}R[V]\sigma,\ S_0 \,\rangle\Downarrow_{(h,s)}.$$

Thus assume

$$\langle\, \Gamma_0,\ ({}^{w \curlyvee X}N)\sigma,\ S_0 \,\rangle\Downarrow_{(h,s)}$$

i.e., that

$$\begin{array}{ll}
 & \langle\, \Gamma_0,\ ({}^{w \curlyvee X}N)\sigma,\ S_0 \,\rangle \\
\equiv & \langle\, \Gamma_0,\ ({}^{w}\mathsf{case\ true\ of\ } \{\mathsf{true} \to {}^{X}N\})\sigma,\ S_0 \,\rangle \\
\to_{(h,s)} & \langle\, \Gamma_0,\ \mathsf{true},\ {}^{w}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{\mathsf{true} \to ({}^{X}N)\sigma\} : S_0 \,\rangle \\
\gg & \langle\, \Gamma_1,\ \mathsf{true},\ {}^{w}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{\mathsf{true} \to ({}^{X}N)\sigma\} : S_1 \,\rangle \\
\to_{(h,s)} & \langle\, \Gamma_1,\ ({}^{X}N)\sigma,\ S_1 \,\rangle \\
\gg & \langle\, \Gamma_2,\ ({}^{X}N)\sigma,\ S_2 \,\rangle \\
\equiv & \langle\, \Gamma_2,\ (\mathsf{let}\ \{\vec{y} = \vec{x}\}\ \mathsf{in}\ N)\sigma,\ S_2 \,\rangle \qquad \text{where } X = \{\vec{x}\} \text{ and } \vec{y} \text{ fresh} \\
\to_{(h,s)} & \langle\, \Gamma_2\{\vec{y} = \vec{x}\sigma\},\ N\sigma,\ S_2 \,\rangle \\
\gg & \langle\, \Gamma_3\Delta_3,\ N\sigma,\ S_3 \,\rangle\Downarrow_{(h,s)} \qquad \text{where } \Gamma_3 \subseteq \Gamma_2 \text{ and } \Delta_3 \subseteq \{\vec{y} = \vec{x}\sigma\}
\end{array}$$

Since Δ_3 is garbage in $\langle\, \Gamma_3\Delta_3,\ N\sigma,\ S_3 \,\rangle$ we may conclude by Lemma 8.1 that

$$\langle\, \Gamma_3,\ N\sigma,\ S_3 \,\rangle\Downarrow_{(h,s)}.$$

It follows by the definition of transitions and garbage collection, using $\mathsf{FV}(R[V]) = \mathsf{FV}({}^{X}N)$, that

$$\begin{array}{ll}
 & \langle\, \Gamma_0,\ {}^{w}R[V]\sigma,\ S_0 \,\rangle \\
\to & \langle\, \Gamma_0,\ V\sigma,\ {}^{w}R\sigma : S_0 \,\rangle \\
\gg & \langle\, \Gamma_1,\ V\sigma,\ {}^{w}R\sigma : S_1 \,\rangle \\
\gg & \langle\, \Gamma_2,\ V\sigma,\ {}^{w}R\sigma : S_2 \,\rangle \\
\to & \langle\, \Gamma_2,\ N\sigma,\ S_2 \,\rangle \\
\gg & \langle\, \Gamma_3,\ N\sigma,\ S_2 \,\rangle\Downarrow_{(h,s)}.
\end{array}$$

It remains to show that the configurations in the transition sequence uses at most (h, s) space. For the first configuration we have

$$|\langle\, \Gamma_0,\ {}^{w}R[V]\sigma,\ S_0 \,\rangle| = |\langle\, \Gamma_0,\ ({}^{w \curlyvee X}N)\sigma,\ S_0 \,\rangle| \leq (h, s).$$

For the second configuration,

$$|\langle\, \Gamma_2,\ V\sigma,\ {}^{w}R\sigma : S_2 \,\rangle| \leq |\langle\, \Gamma_1,\ V\sigma,\ {}^{w}R\sigma : S_1 \,\rangle|$$

since $\langle\, \Gamma_1,\ V\sigma,\ {}^{w}R\sigma : S_1 \,\rangle \gtrdot \langle\, \Gamma_2,\ V\sigma,\ {}^{w}R\sigma : S_2 \,\rangle$ and

$$|\langle\, \Gamma_1,\ V\sigma,\ {}^{w}R\sigma : S_1 \,\rangle| = |\langle\, \Gamma_1,\ {}^{w}\mathsf{true},\ {}^{w}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{\mathsf{true} \to ({}^{X}N)\sigma\} : S_1 \,\rangle| \leq (h, s).$$

Thus

$$\langle\, \Gamma_0,\ {}^{w}R[V]\sigma,\ S_0 \,\rangle{\Downarrow_{(h,s)}}$$

as required. □

8.2 Proof of *let-R*

In this section we present the proof of *let-R*:

$$\mathsf{let}\ \Gamma\ \mathsf{in}\ {}^{w}R[M] \lessapprox \gtrsim {}^{w}R[\mathsf{let}\ \Gamma\ \mathsf{in}\ M] \qquad \text{if } \operatorname{dom}\Gamma \subseteq \mathsf{FV}(M) \qquad (\textit{let-R})$$

Let us comment on the side condition

$$\text{if } \operatorname{dom}\Gamma \subseteq \mathsf{FV}(M).$$

It is needed in the right-way improvement: Imagine that the side condition was not there. Then when we compute with $\mathsf{let}\ \Gamma\ \mathsf{in}\ {}^{w}R[M]$ parts of Γ could possibly be garbage collected immediately after its allocation which in turn can lead to that update markers in the stack may be garbage collected. So the stack could shrink arbitrary much *before* ${}^{w}R$ is pushed on to the stack. But when we compute with ${}^{w}R[\mathsf{let}\ \Gamma\ \mathsf{in}\ M]$, the corresponding garbage collection cannot take place until *after* ${}^{w}R$ is pushed on to the stack. If this happens to be at the peak of space use the second computation would take up w more stack units. Except for this subtlety the context lemma makes the law very easy to establish. We will only show the right-way improvement. The left-way improvement follows in an almost identical manner.

Proof. Assume the side condition, i.e., that

$$\operatorname{dom}\Gamma \subseteq \mathsf{FV}(M).$$

Because of the standard free-variable convention [Bar81] we know that the free variables in any instance of the law are distinct from the bound variables so

$$\mathsf{FV}(R) \cap \operatorname{dom}\Gamma = \emptyset.$$

Thus

$$\mathsf{FV}(\mathsf{let}\ \Gamma\ \mathsf{in}\ R[M]) = \mathsf{FV}(R[\mathsf{let}\ \Gamma\ \mathsf{in}\ M]).$$

By the context lemma it is enough to show that for all Γ_0, S_0 and σ,

$$\begin{array}{l}\langle\, \Gamma_0,\ (\mathsf{let}\ \Gamma\ \mathsf{in}\ R[M])\sigma,\ S_0\,\rangle\Downarrow_{(h,s)}\\ \Longrightarrow\\ \langle\, \Gamma_0,\ R[\mathsf{let}\ \Gamma\ \mathsf{in}\ M]\sigma,\ S_0\,\rangle\Downarrow_{(h,s)}\end{array}$$

Thus assume

$$\langle\, \Gamma_0,\ (\mathsf{let}\ \Gamma\ \mathsf{in}\ R[M])\sigma,\ S_0\,\rangle\Downarrow_{(h,s)}$$

ie that

$$\begin{array}{ll} & \langle\, \Gamma_0,\ (\mathsf{let}\ \Gamma\ \mathsf{in}\ R[M])\sigma,\ S_0\,\rangle\\ \rightarrow_{(h,s)} & \langle\, \Gamma_0\Gamma\sigma,\ R[M]\sigma,\ S_0\,\rangle\\ \gg & \langle\, \Gamma_1\Gamma\sigma,\ R[M]\sigma,\ S_1\,\rangle\\ \rightarrow_{(h,s)} & \langle\, \Gamma_1\Gamma\sigma,\ M\sigma,\ R\sigma : S_1\,\rangle\\ \gg & \langle\, \Gamma_2\Gamma\sigma,\ M\sigma,\ R\sigma : S_2\,\rangle\Downarrow_{(h,s)}.\end{array}$$

We have assumed without loss of generality that $\mathrm{dom}\,\Gamma$ do not clash with the variables bound by the configuration and that σ acts as the identity on $\mathrm{dom}\,\Gamma$. The last assumption means in particular that $\mathrm{dom}\,\Gamma \subseteq \mathsf{FV}(M\sigma)$ since $\mathrm{dom}\,\Gamma \subseteq \mathsf{FV}(M)$. This is what guarantees that $\Gamma\sigma$ is live in $\langle\, \Gamma_1\Gamma\sigma,\ R[M]\sigma,\ S_1\,\rangle$ and $\langle\, \Gamma_2\Gamma\sigma,\ M\sigma,\ R\sigma : S_2\,\rangle$. It follows immediately by the definition of transitions and garbage collection, using $\mathsf{FV}(\mathsf{let}\ \Gamma\ \mathsf{in}\ R[M]) = \mathsf{FV}(R[\mathsf{let}\ \Gamma\ \mathsf{in}\ M])$, that

$$\begin{array}{ll} & \langle\, \Gamma_0,\ R[\mathsf{let}\ \Gamma\ \mathsf{in}\ M]\sigma,\ S_0\,\rangle\\ \rightarrow & \langle\, \Gamma_0,\ (\mathsf{let}\ \Gamma\ \mathsf{in}\ M)\sigma,\ R\sigma : S_0\,\rangle\\ \gg & \langle\, \Gamma_1,\ (\mathsf{let}\ \Gamma\ \mathsf{in}\ M)\sigma,\ R\sigma : S_1\,\rangle\\ \gg & \langle\, \Gamma_2,\ (\mathsf{let}\ \Gamma\ \mathsf{in}\ M)\sigma,\ R\sigma : S_2\,\rangle\\ \rightarrow & \langle\, \Gamma_2\Gamma\sigma,\ M\sigma,\ R\sigma : S_2\,\rangle\\ \gg & \langle\, \Gamma_2\Gamma\sigma,\ M\sigma,\ R\sigma : S_2\,\rangle\Downarrow_{(h,s)}.\end{array}$$

It remains to show that the configurations in the transition sequence uses at most (h, s) space. For the first configuration we have

$$|\langle\, \Gamma_0,\ R[\mathsf{let}\ \Gamma\ \mathsf{in}\ M]\sigma,\ S_0\,\rangle| = |\langle\, \Gamma_0,\ (\mathsf{let}\ \Gamma\ \mathsf{in}\ R[M])\sigma,\ S_0\,\rangle| \leq (h, s),$$

and for the second configuration

$$|\langle\, \Gamma_2,\ (\mathsf{let}\ \Gamma\ \mathsf{in}\ M)\sigma,\ R\sigma : S_2\,\rangle| < |\langle\, \Gamma_2\Gamma\sigma,\ M\sigma,\ R\sigma : S_2\,\rangle| \leq (h, s).$$

Thus

$$\langle\, \Gamma_0,\ R[\mathsf{let}\ \Gamma\ \mathsf{in}\ M]\sigma,\ S_0\,\rangle\Downarrow_{(h,s)}$$

as required. □

8.3 Proof of *R-case*

In this section we present the proof of *R-case*:

$$^{w}R[^{v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}] \mathrel{\lessapprox\gtrapprox} {}^{w+v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to {}^{w}R[N_i]\} \qquad (R\text{-}case)$$

The proof is more complicated than the proofs of *reduction* and *let-R* from the previous sections. It is more involved because when the two terms are placed in the evaluation position of a configuration the respective configurations does not evaluate to the same configuration up to garbage within a few steps. It is not until after the evaluation of M has finished that the two computations lead to identical configurations. To deal with this situation we need to run the computation just until the case alternatives are to be popped off the stack. Recall the Uniform Computation Lemma from Section 7.2 which we used to prove the context lemma. It allowed us to unplug a subterm and run the computation until it depended on the subterm, or until termination if the computation was independent of the subterm in question. Here, we need to unplug parts of the stack and run the computation until it depends on this part of the stack. In their work on time improvement [MS99b], Moran and Sands introduced the notion of Open Uniform Computation where they can take out and put back parts of the heap and parts of the stack and they have an Open Uniform Computation Lemma similar to, but more powerful than, our Uniform Computation Lemma. They used this technique to establish *R-case* for strong time improvement. However, their technique is not directly applicable to a semantics with garbage collection: during computation garbage collection may remove bindings and update markers so if we were to take them out it would interfere with garbage collection. Also, if we, without taking extra care, take out parts of the heap and the stack we could remove references to bindings which could lead to too early garbage collection.

The solution is to introduce the notion of a stack hole which may be plugged with a substack which must not contain any update markers. We will let ψ range over stack hole variables. Analogously to hole variables, each stack hole variables have an associated arity n and an occurrence of the stack hole variable is applied to a vector of variables of length n. Each stack hole variable is also decorated with a weight which specifies how much stack space it should account for. Stack holes may be plugged with stack plugs which is of the form $\Lambda\vec{x}.S$ where S is a stack without any update marker. We will use Ψ to range over stack plugs. To plug $\Lambda\vec{x}.S$ into ${}^{w}\psi\,\vec{y}$ we require that the sum of the weights in S equals w. If so the result is $S[\vec{y}/\vec{x}]$. We will leave this condition on stack plugs implicit in the rest of this paper.

We tacitly lift the semantics to computing with configurations with stack holes. As with term holes, filling a stack hole commutes with computation.

Lemma 8.2
For every closed stack plug Ψ,

if $\langle\, \Gamma,\ M,\ {}^{w}\psi\,\vec{x}\,S\,\rangle \rightarrowtriangle^{m}_{(h,s)} \langle\, \Delta,\ V,\ {}^{w}\psi\,\vec{x}\,T\,\rangle$ then

$$\langle\, \Gamma,\ M,\ {}^{w}\psi\,\vec{x}\,S\,\rangle[\Psi/\psi] \rightarrowtriangle^{m}_{(h,s)} \langle\, \Delta,\ V,\ {}^{w}\psi\,\vec{x}\,T\,\rangle[\Psi/\psi],$$

PROOF. By induction over the length of the computation. □

We also have a Uniform Computation Lemma for unplugging stack holes.

Lemma 8.3 (Uniform Computation)
For every closed linear stack plug Ψ, if $\langle\, \Gamma,\ M,\ {}^{w}\psi\,\vec{x}\,S\,\rangle[\Psi/\psi]\Downarrow^{n}_{(h,s)}$ then there exists m, Δ, V, T such that

- $\langle\, \Gamma,\ M,\ {}^{w}\psi\,\vec{x}\,S\,\rangle \rightarrowtriangle^{m}_{(h,s)} \langle\, \Delta,\ V,\ {}^{w}\psi\,\vec{x}\,T\,\rangle$,
- $\langle\, \Delta,\ V,\ {}^{w}\psi\,\vec{x}\,T\,\rangle[\Psi/\psi]\Downarrow^{n-m}_{(h,s)}$, and
- T can be obtained from S by removing zero or more update markers.

PROOF. By induction over the length of the computation. □

We are now ready to show *R-case*. We will only show the right-way improvement. The left-way improvement follows in an almost identical manner.

PROOF. Because of the standard free-variable convention [Bar81] we know that the free variables in any instance of the law are distinct from the bound variables so

$$\mathsf{FV}(R) \cap \operatorname{dom} pat_{i} = \emptyset.$$

Thus

$$\mathsf{FV}({}^{w}R[{}^{v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_{i} \rightarrow N_{i}\}]) = \mathsf{FV}({}^{w+v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_{i} \rightarrow {}^{w}R[N_{i}]\}).$$

By the context lemma it is enough to show that for all Γ_0, S_0 and σ,

$$\begin{array}{l}\langle\, \Gamma_0,\ {}^{w}R[{}^{v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_{i} \rightarrow N_{i}\}]\sigma,\ S_0\,\rangle\Downarrow_{(h,s)} \\ \Longrightarrow \\ \langle\, \Gamma_0,\ ({}^{w+v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_{i} \rightarrow {}^{w}R[N_{i}]\})\sigma,\ S_0\,\rangle\Downarrow_{(h,s)}\end{array}$$

Thus assume

$$\langle\, \Gamma_0,\ {}^{w}R[{}^{v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_{i} \rightarrow N_{i}\}]\sigma,\ S_0\,\rangle\Downarrow_{(h,s)},$$

i.e., that

$$\begin{array}{ll} & \langle\, \Gamma_0,\ {}^{w}R[{}^{v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_{i} \rightarrow N_{i}\}]\sigma,\ S_0\,\rangle \\ \rightarrow_{(h,s)} & \langle\, \Gamma_0,\ ({}^{v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_{i} \rightarrow N_{i}\})\sigma,\ {}^{w}R\sigma : S_0\,\rangle \\ \gtrdot & \langle\, \Gamma_1,\ ({}^{v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_{i} \rightarrow N_{i}\})\sigma,\ {}^{w}R\sigma : S_1\,\rangle \\ \rightarrow_{(h,s)} & \langle\, \Gamma_1,\ M\sigma,\ {}^{v}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_{i} \rightarrow N_{i}\sigma\} : {}^{w}R\sigma : S_1\,\rangle \\ \gtrdot & \langle\, \Gamma_2,\ M\sigma,\ {}^{v}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_{i} \rightarrow N_{i}\sigma\} : {}^{w}R\sigma : S_2\,\rangle\Downarrow_{(h,s)}.\end{array}$$

We have assumed, without loss of generality, that σ acts as the identity on the variables bound by the patterns. Let $\vec{x}$ be a vector with the free variables of ${}^{v}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to N_i\sigma\}$ and ${}^{w}R\sigma$. Then $\Lambda\vec{x}.{}^{v}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to N_i\sigma\} : {}^{w}R\sigma$ is a closed linear stack plug and

$$\begin{array}{ll} & \langle\, \Gamma_2,\ M\sigma,\ {}^{w+v}\psi\,\vec{x}\,S_2\,\rangle[\Lambda\vec{x}.{}^{v}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to N_i\sigma\} : {}^{w}R\sigma/_{\psi}] \\ \equiv & \langle\, \Gamma_2,\ M\sigma,\ {}^{v}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to N_i\sigma\} : {}^{w}R\sigma : S_2\,\rangle\Downarrow_{(h,s)} \end{array}$$

so it follows by the Uniform Computation Lemma that

$$\langle\, \Gamma_2,\ M\sigma,\ {}^{w+v}\psi\,\vec{x}\,S_2\,\rangle \to^{n}_{(h,s)} \langle\, \Gamma_3,\ V,\ {}^{w+v}\psi\,\vec{x}\,S_3\,\rangle$$

and

$$\langle\, \Gamma_3,\ V,\ {}^{w+v}\psi\,\vec{x}\,S_3\,\rangle[\Lambda\vec{x}.{}^{v}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to N_i\sigma\} : {}^{w}R\sigma/_{\psi}]\Downarrow_{(h,s)}.$$

From the latter it follows directly that V matches one of the patterns pat_j and that

$$\begin{array}{ll} & \langle\, \Gamma_3,\ V,\ {}^{v}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to N_i\sigma\} : {}^{w}R\sigma : S_3\,\rangle \\ \to_{(h,s)} & \langle\, \Gamma_3,\ N_j\sigma\sigma_j,\ {}^{w}R\sigma : S_3\,\rangle \\ \gg & \langle\, \Gamma_4,\ N_j\sigma\sigma_j,\ {}^{w}R\sigma : S_4\,\rangle\Downarrow_{(h,s)}. \end{array}$$

Apart from the step marked with $(*)$ it follows, by the definition of transitions and garbage collection using

$$\mathsf{FV}({}^{w}R[{}^{v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}]) = \mathsf{FV}({}^{w+v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to {}^{w}R[N_i]\}),$$

that

$$\begin{array}{lll} & \langle\, \Gamma_0,\ ({}^{w+v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to {}^{w}R[N_i]\})\sigma,\ S_0\,\rangle & \\ \to & \langle\, \Gamma_0,\ M\sigma,\ {}^{w+v}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to {}^{w}R[N_i]\sigma\} : S_0\,\rangle & \\ \gg & \langle\, \Gamma_1,\ M\sigma,\ {}^{w+v}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to {}^{w}R[N_i]\sigma\} : S_1\,\rangle & \\ \gg & \langle\, \Gamma_2,\ M\sigma,\ {}^{w+v}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to {}^{w}R[N_i]\sigma\} : S_2\,\rangle & \\ \to^{n}_{(h,s)} & \langle\, \Gamma_3,\ V,\ {}^{w+v}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to {}^{w}R[N_i]\sigma\} : S_3\,\rangle & (*) \\ \to & \langle\, \Gamma_3,\ {}^{w}R[N_j]\sigma\sigma_j,\ S_3\,\rangle & \\ \gg & \langle\, \Gamma_4,\ {}^{w}R[N_j]\sigma\sigma_j,\ S_4\,\rangle & \\ \to & \langle\, \Gamma_4,\ N_j\sigma\sigma_j,\ {}^{w}R\sigma : S_4\,\rangle & \\ \gg & \langle\, \Gamma_4,\ N_j\sigma\sigma_j,\ {}^{w}R\sigma : S_4\,\rangle\Downarrow_{(h,s)}. & \end{array}$$

The step marked with $(*)$ follows by filling the hole in $\langle\, \Gamma_2,\ M\sigma,\ {}^{w+v}\psi\,\vec{x}S_2\,\rangle \to^{n}_{(h,s)} \langle\, \Gamma_3,\ V,\ {}^{w+v}\psi\,\vec{x}S_3\,\rangle$ with $\Lambda\vec{x}.{}^{w+v}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to {}^{w}R[N_i]\sigma\}$. It remains to show that the configurations in the transition sequence uses at most (h,s) space. For the first configuration we have

$$\begin{array}{l} |\langle\, \Gamma_0,\ ({}^{w+v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to {}^{w}R[N_i]\})\sigma,\ S_0\,\rangle| \\ \qquad\qquad = |\langle\, \Gamma_0,\ {}^{w}R[{}^{v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}]\sigma,\ S_0\,\rangle| \leq (h,s), \end{array}$$

for the second configuration

$$|\langle\, \Gamma_3,\ V,\ {}^{w+v}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to {}^{w}R[N_i]\sigma\} : S_3\,\rangle| \\ = |\langle\, \Gamma_3,\ V,\ {}^{v}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to N_i\sigma\}) : {}^{w}R\sigma : S_3\,\rangle| \leq (h,s),$$

and for the third configuration

$$|\langle\, \Gamma_4,\ {}^{w}R[N_j]\sigma\sigma_j,\ S_4\,\rangle| < |\langle\, \Gamma_4,\ N_j\sigma\sigma_j,\ {}^{w}R\sigma : S_4\,\rangle| \leq (h,s).$$

Thus

$$\langle\, \Gamma_0,\ ({}^{w+v}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to {}^{w}R[N_i]\})\sigma,\ S_0\,\rangle \Downarrow_{(h,s)}$$

as required. □

8.4 Proof of *let-alts*

In this section we present the proof of *let-alts*:

$$\mathsf{let}\ \Gamma\ \mathsf{in}\ {}^{w}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Delta\ \mathsf{in}\ N_i\} \\ \lessapprox \mathsf{let}\ \Delta\ \mathsf{in}\ {}^{w}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Gamma\ \mathsf{in}\ N_i\}, \\ \text{if } \operatorname{dom}\Gamma \cup \operatorname{dom}\Delta \subseteq \mathsf{FV}(N_i), \text{ and } |\Gamma| = |\Delta|. \qquad (\textit{let-alts})$$

The proof is similar to the proof of *R-case* in that it uses stack holes but there are some additional complications: We need to argue that we can swap the allocation of Γ and Δ. The argument can broken into two steps. First we need to argue that the bindings in Γ are live and untouched after the evaluation of M. That they are live should be intuitively obvious: because of the side condition $\operatorname{dom}\Gamma \subseteq \mathsf{FV}(N_i)$ they cannot be garbage collected. That they are untouched follows from that $\operatorname{dom}\Gamma \cap \mathsf{FV}(M) = \emptyset$ (implicit by the free-variable convention) so they can't be needed in the computation of M. The following technical Lemma is needed in the proof to make this informal argument precise.

Lemma 8.4
If

- $\langle\, \Gamma\Delta,\ M,\ {}^{w}\psi\,\vec{x}\,S\,\rangle \twoheadrightarrow^{m}_{(h,s)} \langle\, \Gamma',\ M',\ {}^{w}\psi\,\vec{x}\,S'\,\rangle$,
- $\operatorname{dom}\Delta \subseteq \{\vec{x}\}$ and
- $\operatorname{dom}\Delta \cap \mathsf{FV}(\Gamma, M, S) = \emptyset$

then there exists $\langle\, \Gamma''\Delta,\ M'',\ {}^{w}\psi\,\vec{x}\,S''\,\rangle$ such that

$$\langle\, \Gamma',\ M',\ {}^{w}\psi\,\vec{x}\,S'\,\rangle \equiv \langle\, \Gamma''\Delta,\ M'',\ {}^{w}\psi\,\vec{x}\,S''\,\rangle.$$

Proof. By induction over the length of the computation. □

The next step that needs to be argued by the proof is that since the bindings in Γ are live and untouched after the evaluation of M we can delay the allocation to the branches of the case and instead allocate Δ earlier. The following technical Lemma is needed to carry out this step of the proof.

Lemma 8.5
If

- $\langle\, \Gamma\Delta,\ M,\ {}^{w}\psi\, \vec{x}\, S \,\rangle \twoheadrightarrow^{m}_{(h,s)} \langle\, \Gamma'\Delta,\ M',\ {}^{w}\psi\, \vec{x}\, S' \,\rangle$,
- $\operatorname{dom}\Delta \cap \mathsf{FV}(\Gamma, M, S) = \emptyset$
- $|\Delta| = |\Delta'|$,
- $(\mathsf{FV}(\Delta) \cup \{\vec{x}\}) \setminus \operatorname{dom}\Delta = (\mathsf{FV}(\Delta') \cup \{\vec{y}\}) \setminus \operatorname{dom}\Delta'$ and
- $\langle\, \Gamma\Delta',\ M,\ {}^{ww}\psi'\, \vec{y}\, S \,\rangle$ is a well formed configuration,

then

$$\langle\, \Gamma\Delta',\ M,\ {}^{ww}\psi'\, \vec{y}\, S \,\rangle \twoheadrightarrow^{m}_{(h,s)} \langle\, \Gamma'\Delta',\ M',\ {}^{ww}\psi'\, \vec{y}\, S' \,\rangle.$$

Proof. By induction over the length of the computation. □

With these technical lemmas at hand we can proceed with the proof of *let-alts*. The law is symmetric so it suffices to prove one direction.

Proof. Assume the side conditions i.e., that

$$\operatorname{dom}\Gamma \cup \operatorname{dom}\Delta \subseteq \mathsf{FV}(N_i),$$

and

$$|\Gamma| = |\Delta|.$$

Because of the standard free-variable convention [Bar81] we know that all bound variables in in any instance of the law are distinct, and that they are disjoint from the free variables. We will use this silently throughout the proof. A first consequence is that

$$\begin{aligned}\mathsf{FV}(\mathsf{let}\ \Gamma\ \mathsf{in}\ {}^{w}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Delta\ \mathsf{in}\ N_i\}) \\ = \mathsf{FV}(\mathsf{let}\ \Delta\ \mathsf{in}\ {}^{w}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Gamma\ \mathsf{in}\ N_i\}).\end{aligned}$$

By the context lemma it is enough to show that for all Γ_0, S_0 and σ,

$$\begin{aligned}&\langle\, \Gamma_0,\ (\mathsf{let}\ \Gamma\ \mathsf{in}\ {}^{w}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Delta\ \mathsf{in}\ N_i\})\sigma,\ S_0 \,\rangle\Downarrow_{(h,s)} \\ &\Longrightarrow \\ &\langle\, \Gamma_0,\ (\mathsf{let}\ \Delta\ \mathsf{in}\ {}^{w}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Gamma\ \mathsf{in}\ N_i\})\sigma,\ S_0 \,\rangle\Downarrow_{(h,s)}\end{aligned}$$

Thus assume

$$\langle\, \Gamma_0,\ (\mathsf{let}\ \Gamma\ \mathsf{in}\ {}^w\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Delta\ \mathsf{in}\ N_i\})\sigma,\ S_0 \,\rangle{\Downarrow_{(h,s)}}$$

i.e., that

$$\begin{array}{ll} & \langle\, \Gamma_0,\ (\mathsf{let}\ \Gamma\ \mathsf{in}\ {}^w\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Delta\ \mathsf{in}\ N_i\})\sigma,\ S_0 \,\rangle \\ \to_{(h,s)} & \langle\, \Gamma_0\Gamma\sigma,\ ({}^w\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Delta\ \mathsf{in}\ N_i\})\sigma,\ S_0 \,\rangle \\ \gtrdot & \langle\, \Gamma_1\Gamma\sigma,\ ({}^w\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Delta\ \mathsf{in}\ N_i\})\sigma,\ S_1 \,\rangle \\ \to_{(h,s)} & \langle\, \Gamma_1\Gamma\sigma,\ M\sigma,\ {}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Delta\ \mathsf{in}\ N_i)\sigma\} : S_1 \,\rangle \\ \gtrdot & \langle\, \Gamma_2\Gamma\sigma,\ M\sigma,\ {}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Delta\ \mathsf{in}\ N_i)\sigma\} : S_2 \,\rangle{\Downarrow_{(h,s)}}. \end{array}$$

We have assumed, without loss of generality, that σ acts as the identity on the variables bound by the patterns and the lets. Let $\vec{x}$ be a vector with the free variables of ${}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Delta\ \mathsf{in}\ N_i)\sigma\}$. Then

$$\Lambda\vec{x}.{}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Delta\ \mathsf{in}\ N_i)\sigma\}$$

is a closed linear stack plug and

$$\begin{array}{ll} & \langle\, \Gamma_2\Gamma\sigma,\ M\sigma,\ {}^w\psi\,\vec{x}\,S_2 \,\rangle[\Lambda\vec{x}.{}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Delta\ \mathsf{in}\ N_i)\sigma\}/\psi] \\ \equiv & \langle\, \Gamma_2\Gamma\sigma,\ M\sigma,\ {}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Delta\ \mathsf{in}\ N_i)\sigma\} : S_2 \,\rangle{\Downarrow_{(h,s)}} \end{array}$$

so it follows by the Uniform Computation Lemma that

$$\langle\, \Gamma_2\Gamma\sigma,\ M\sigma,\ {}^w\psi\,\vec{x}\,S_2 \,\rangle \twoheadrightarrow^n_{(h,s)} \langle\, \Delta_0,\ V_0,\ {}^w\psi\,\vec{x}\,T_0 \,\rangle$$

and

$$\langle\, \Delta_0,\ V_0,\ {}^w\psi\,\vec{x} : T_0 \,\rangle[\Lambda\vec{x}.{}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Delta\ \mathsf{in}\ N_i)\sigma\}/\psi]{\Downarrow_{(h,s)}}.$$

From the first side condition of the law we know that $\mathrm{dom}\,\Gamma \subseteq \mathsf{FV}(N_i)$ and since $\mathrm{dom}\,\Gamma \cap \mathrm{dom}\,\Delta = \emptyset$ and $\mathrm{dom}\,\Gamma \cap \mathrm{dom}\,pat_i = \emptyset$ (the free variable convention) we have that

$$\mathrm{dom}\,\Gamma\sigma \subseteq \mathsf{FV}({}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Delta\ \mathsf{in}\ N_i)\sigma\}) = \{\vec{x}\}.$$

Also, from the free variable convention

$$\mathrm{dom}\,\Gamma\sigma \cap \mathsf{FV}(\Gamma_2, M\sigma, S_2) = \emptyset.$$

From these facts and

$$\langle\, \Gamma_2\Gamma\sigma,\ M\sigma,\ {}^w\psi\,\vec{x}\,S_2 \,\rangle \twoheadrightarrow^n_{(h,s)} \langle\, \Delta_0,\ V_0,\ {}^w\psi\,\vec{x}\,T_0 \,\rangle$$

we know that $\Gamma\sigma$ must be a part of Δ_0, i.e., by Lemma 8.4,

$$\langle\, \Delta_0,\ V_0,\ {}^w\psi\,\vec{x} : T_0 \,\rangle \equiv \langle\, \Gamma_3\Gamma\sigma,\ V,\ {}^w\psi\,\vec{x}\,S_3 \,\rangle$$

for some Γ_3, V and S_3. Thus we have

$$\langle\, \Gamma_2\Gamma\sigma,\ M\sigma,\ {}^w\psi\,\vec{x}\,S_2\,\rangle \twoheadrightarrow^n_{(h,s)} \langle\, \Gamma_3\Gamma\sigma,\ V,\ {}^w\psi\,\vec{x}\,S_3\,\rangle$$

and

$$\begin{array}{ll} & \langle\, \Delta_0,\ V_0,\ {}^w\psi\,\vec{x} : T_0\,\rangle[\Lambda\vec{x}.{}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Delta\ \mathsf{in}\ N_i)\sigma\}/_\psi] \\ \equiv & \langle\, \Gamma_3\Gamma\sigma,\ V,\ {}^w\psi\,\vec{x}\,S_3\,\rangle[\Lambda\vec{x}.{}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Delta\ \mathsf{in}\ N_i)\sigma\}/_\psi] \\ \equiv & \langle\, \Gamma_3\Gamma\sigma,\ V,\ {}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Delta\ \mathsf{in}\ N_i)\sigma\} : S_3\,\rangle\Downarrow_{(h,s)}. \end{array}$$

From the latter it follows directly that V matches one of the patterns pat_j and that

$$\begin{array}{ll} & \langle\, \Gamma_3\Gamma\sigma,\ V,\ {}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Delta\ \mathsf{in}\ N_i)\sigma\} : S_3\,\rangle \\ \to_{(h,s)} & \langle\, \Gamma_3\Gamma\sigma,\ (\mathsf{let}\ \Delta\ \mathsf{in}\ N_i)\sigma\sigma_j,\ S_3\,\rangle \\ \gg & \langle\, \Gamma_4\Gamma\sigma,\ (\mathsf{let}\ \Delta\ \mathsf{in}\ N_i)\sigma\sigma_j,\ S_4\,\rangle \\ \to_{(h,s)} & \langle\, \Gamma_4\Gamma\sigma\Delta\sigma,\ N_i\sigma\sigma_j,\ S_4\,\rangle \\ \gg & \langle\, \Gamma_5\Gamma\sigma\Delta\sigma,\ N_i\sigma\sigma_j,\ S_5\,\rangle\Downarrow_{(h,s)}. \end{array}$$

Apart from the step marked with $(*)$ it follows, by the definition of transitions and garbage collection using

$$\begin{array}{l} \mathsf{FV}(\mathsf{let}\ \Gamma\ \mathsf{in}\ {}^w\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Delta\ \mathsf{in}\ N_i\}) \\ \qquad\qquad = \mathsf{FV}(\mathsf{let}\ \Delta\ \mathsf{in}\ {}^w\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Gamma\ \mathsf{in}\ N_i\}), \end{array}$$

that

$$\begin{array}{lll} & \langle\, \Gamma_0,\ (\mathsf{let}\ \Delta\ \mathsf{in}\ {}^w\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Gamma\ \mathsf{in}\ N_i\})\sigma,\ S_0\,\rangle & \\ \to & \langle\, \Gamma_0\Delta\sigma,\ ({}^w\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Gamma\ \mathsf{in}\ N_i\})\sigma,\ S_0\,\rangle & \\ \gg & \langle\, \Gamma_1\Delta\sigma,\ ({}^w\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Gamma\ \mathsf{in}\ N_i\})\sigma,\ S_1\,\rangle & \\ \to & \langle\, \Gamma_1\Delta\sigma,\ M\sigma,\ {}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Gamma\ \mathsf{in}\ N_i)\sigma\} : S_1\,\rangle & \\ \gg & \langle\, \Gamma_2\Delta\sigma,\ M\sigma,\ {}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Gamma\ \mathsf{in}\ N_i)\sigma\} : S_2\,\rangle & \\ \twoheadrightarrow^n_{(h,s)} & \langle\, \Gamma_3\Delta\sigma,\ V,\ {}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Gamma\ \mathsf{in}\ N_i)\sigma\} : S_3\,\rangle & (*) \\ \to & \langle\, \Gamma_3\Delta\sigma,\ (\mathsf{let}\ \Gamma\ \mathsf{in}\ N_i)\sigma\sigma_j,\ S_3\,\rangle & \\ \gg & \langle\, \Gamma_4\Delta\sigma,\ (\mathsf{let}\ \Gamma\ \mathsf{in}\ N_i)\sigma\sigma_j,\ S_4\,\rangle & \\ \to & \langle\, \Gamma_4\Delta\sigma\Gamma\sigma,\ N_i\sigma\sigma_j,\ S_4\,\rangle & \\ \equiv & \langle\, \Gamma_4\Gamma\sigma\Delta\sigma,\ N_i\sigma\sigma_j,\ S_4\,\rangle & \\ \gg & \langle\, \Gamma_5\Gamma\sigma\Delta\sigma,\ N_i\sigma\sigma_j,\ S_5\,\rangle\Downarrow_{(h,s)}. & \end{array}$$

The step marked with $(*)$ is a bit involved. Recall that we showed that

$$\langle\, \Gamma_2\Gamma\sigma,\ M\sigma,\ {}^w\psi\,\vec{x}\,S_2\,\rangle \twoheadrightarrow^n_{(h,s)} \langle\, \Gamma_3\Gamma\sigma,\ V,\ {}^w\psi\,\vec{x}\,S_3\,\rangle.$$

Let $\vec{y}$ be the free variables of ${}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Gamma\ \mathsf{in}\ N_i)\sigma\}$. The second side condition of the law specifies that $|\Gamma| = |\Delta|$, i.e., that Γ and Δ takes up the same amount of space, and we have that

$$\begin{array}{ll} & (\mathsf{FV}(\Gamma\sigma) \cup \{\vec{x}\}) \setminus \mathrm{dom}\,\Gamma\sigma \\ = & (\mathsf{FV}(\Gamma\sigma) \cup \mathsf{FV}({}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Delta\ \mathsf{in}\ N_i)\sigma\})) \setminus \mathrm{dom}\,\Gamma\sigma \\ = & (\mathsf{FV}(\Delta\sigma) \cup \mathsf{FV}({}^w\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Gamma\ \mathsf{in}\ N_i)\sigma\})) \setminus \mathrm{dom}\,\Delta\sigma \\ = & (\mathsf{FV}(\Delta\sigma) \cup \{\vec{y}\}) \setminus \mathrm{dom}\,\Delta\sigma \end{array}$$

where we have used that the variables bound the lets and the patterns are distinct and disjoint from the free variables. We have already argued that $\operatorname{dom}\Gamma\sigma \cap \mathsf{FV}(\Gamma_2, M\sigma, S_2) = \emptyset$. so from Lemma 8.5 it follows that

$$\langle\, \Gamma_2\Delta\sigma,\ M\sigma,\ {}^{w}\psi'\,\vec{y}\,S_2\,\rangle \twoheadrightarrow^{n}_{(h,s)} \langle\, \Gamma_3\Delta\sigma,\ V,\ {}^{w}\psi'\,\vec{y}\,S_3\,\rangle.$$

Now $(*)$ follows by plugging the hole with $\Lambda\vec{y}.{}^{w}\mathsf{case}\,[\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Gamma\ \mathsf{in}\ N_i)\sigma\}$. It remains to show that the configurations in the transition sequence uses at most (h,s) space. For the first configuration we have

$$\begin{aligned}&|\langle\, \Gamma_0,\ (\mathsf{let}\ \Delta\ \mathsf{in}\ {}^{w}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Gamma\ \mathsf{in}\ N_i\})\sigma,\ S_0\,\rangle| \\ &\qquad = |\langle\, \Gamma_0,\ (\mathsf{let}\ \Gamma\ \mathsf{in}\ {}^{w}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Delta\ \mathsf{in}\ N_i\})\sigma,\ S_0\,\rangle| \le (h,s),\end{aligned}$$

for the second configuration

$$\begin{aligned}&|\langle\, \Gamma_1\Delta\sigma,\ ({}^{w}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Gamma\ \mathsf{in}\ N_i\})\sigma,\ S_1\,\rangle| \\ &\qquad = |\langle\, \Gamma_1\Gamma\sigma,\ ({}^{w}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Delta\ \mathsf{in}\ N_i\})\sigma,\ S_1\,\rangle| \le (h,s),\end{aligned}$$

for the third configuration

$$\begin{aligned}&|\langle\, \Gamma_3\Delta\sigma,\ V,\ {}^{w}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Gamma\ \mathsf{in}\ N_i)\sigma\} : S_3\,\rangle| \\ &\qquad = |\langle\, \Gamma_3\Gamma\sigma,\ V,\ {}^{w}\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{pat_i \to (\mathsf{let}\ \Delta\ \mathsf{in}\ N_i)\sigma\} : S_3\,\rangle| \le (h,s),\end{aligned}$$

and finally for the forth configuration

$$|\langle\, \Gamma_4\Delta\sigma,\ (\mathsf{let}\ \Gamma\ \mathsf{in}\ N_i)\sigma\sigma_j,\ S_4\,\rangle| = |\langle\, \Gamma_4\Gamma\sigma,\ (\mathsf{let}\ \Delta\ \mathsf{in}\ N_i)\sigma\sigma_j,\ S_4\,\rangle| \le (h,s).$$

Thus

$$\langle\, \Gamma_0,\ (\mathsf{let}\ \Delta\ \mathsf{in}\ {}^{w}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ \Gamma\ \mathsf{in}\ N_i\})\sigma,\ S_0\,\rangle\Downarrow_{(h,s)}$$

as required. □

8.5 Proof of *let-let*

In this section we present the proof of *let-let*:

$$\mathsf{let}\ \Gamma\{{}^{v}_{w}x = \mathsf{let}\ \Delta\ \mathsf{in}\ M\}\ \mathsf{in}\ N \lessapprox \mathsf{let}\ \Delta\{{}^{v}_{w}x = \mathsf{let}\ \Gamma\ \mathsf{in}\ M\}\ \mathsf{in}\ N \qquad \text{if } \operatorname{dom}\Gamma \cup \operatorname{dom}\Delta \subseteq \mathsf{FV}(M), \text{ and } |\Gamma| = |\Delta|. \quad (\textit{let-let})$$

To prove the law we need a notion of heap holes. A heap hole is a term hole which occurs uniquely in a configuration in the right hand side of a binding in the heap. That is, ξ is a heap hole in a configuration of the form

$$\langle\, \mathbb{\Gamma}\{{}^{v}_{w}x = \xi\,\vec{x}\},\ \mathbb{M},\ \mathbb{S}\,\rangle$$

if ξ does not occur in $\mathbb{\Gamma}$, $\mathbb{M}$ and $\mathbb{S}$. The key property of heap holes is that they cannot be duplicated by computation. This is in not necessarily the case for term holes. If they occur under an abstraction they may be duplicated if the abstraction is duplicated. We need this property of heap holes to prove the law, because we need to argue that Δ is not allocated repeatedly. That heap holes cannot be duplicated is expressed by the following Uniform Computation Lemma for heap holes.

Lemma 8.6 (Uniform Computation)
For every closed linear plug Ξ, if $\langle\, \Gamma\{x = \xi\, \vec{x}\},\ M,\ S\,\rangle[\Xi/\xi]\Downarrow^n_{(h,s)}$ then

- either $\langle\, \Gamma\{x = \xi\, \vec{x}\},\ M,\ S\,\rangle\Downarrow^n_{(h,s)}$,
- or there exists m, Δ and T such that
  - $\langle\, \Gamma\{x = \xi\, \vec{x}\},\ M,\ S\,\rangle \twoheadrightarrow^m_{(h,s)} \langle\, \Delta,\ \xi\, \vec{x},\ T\,\rangle$
  - $\langle\, \Delta,\ \xi\, \vec{x},\ T\,\rangle[\Xi/\xi]\Downarrow^{n-m}_{(h,s)}$.

Proof. By induction over the length of the computation. □

Just like in the proof of *let-alts* we need to argue that we can swap the allocation of Γ and Δ. The argument is similar to the one for *let-alts* but the technical lemmas given below concerns heap holes rather than stack holes.

Lemma 8.7
If

- $\langle\, \Gamma\Delta\{^v_w x = \xi\, \vec{x}\},\ M,\ S\,\rangle \twoheadrightarrow^m_{(h,s)} \langle\, \Gamma',\ \xi\, \vec{x},\ S'\rangle$,
- $\operatorname{dom}\Delta \subseteq \{\vec{x}\}$ and
- $\operatorname{dom}\Delta \cap \mathsf{FV}(\Gamma, M, S) = \emptyset$

then there exists $\langle\, \Gamma''\Delta,\ \xi\, \vec{x},\ S''\,\rangle$ such that

$$\langle\, \Gamma',\ \xi\, \vec{x},\ S'\,\rangle \equiv \langle\, \Gamma''\Delta,\ \xi\, \vec{x},\ S''\,\rangle.$$

Proof. By induction over the length of the computation. □

Lemma 8.8
If

- $\langle\, \Gamma\Delta\{^v_w x = \xi\, \vec{x}\},\ M,\ S\,\rangle \twoheadrightarrow^m_{(h,s)} \langle\, \Gamma'\Delta,\ \xi\, \vec{x},\ S'\,\rangle$,
- $\operatorname{dom}\Delta \cap \mathsf{FV}(\Gamma, M, S) = \emptyset$
- $|\Delta| = |\Delta'|$,
- $(\mathsf{FV}(\Delta) \cup \{\vec{x}\}) \setminus \operatorname{dom}\Delta = (\mathsf{FV}(\Delta') \cup \{\vec{y}\}) \setminus \operatorname{dom}\Delta'$ and

- $\langle\, \Gamma\Delta'\{^{v}_{w}x = \xi'\,\vec{y}\},\ M,\ S\,\rangle$ is a well formed configuration,

then

$$\langle\, \Gamma\Delta'\{^{v}_{w}x = \xi'\,\vec{y}\},\ M,\ S\,\rangle \twoheadrightarrow^{m}_{(h,s)} \langle\, \Gamma'\Delta',\ \xi'\,\vec{y},\ S'\,\rangle.$$

Proof. By induction over the length of the computation. □

There is an additional aspect of *let-let* that complicates matters. The laws we have proved in the previous sections all have the property that if the two related terms are placed in the evaluation position of a configuration the respective configurations eventually reduces to the same configuration up to garbage. But this not the case for *let-let* because it may happen that the binding for x is never demanded during the computation and then the first computation would never allocate Δ and the second would never allocate Γ. The law is still sound which we argue informally as follows. If x is never demanded then the bindings in Γ and Δ respectively are not demanded either. Since Γ and Δ takes up the same amount of space (the second side condition) and they have the same liveness properties it cannot effect the space behaviour. To make this informal argument precise we need another technical lemma.

Lemma 8.9
If

- $\langle\, \Gamma\Delta\{^{v}_{w}x = \xi\,\vec{x}\},\ M,\ S\,\rangle\Downarrow_{(h,s)}$,
- $|\Delta| = |\Delta'|$,
- $\operatorname{dom}\Delta \cap \mathsf{FV}(\Gamma, M, S) = \emptyset$,
- $\operatorname{dom}\Delta \subseteq \{\vec{x}\}$,
- $(\mathsf{FV}(\Delta) \cup \{\vec{x}\}) \setminus \operatorname{dom}\Delta = (\mathsf{FV}(\Delta') \cup \{\vec{y}\}) \setminus \operatorname{dom}\Delta'$ and
- $\langle\, \Gamma\Delta'\{^{v}_{w}x = \xi'\,\vec{y}\},\ M,\ S\,\rangle$ is a well formed configuration,

then

$$\langle\, \Gamma\Delta'\{^{v}_{w}x = \xi'\,\vec{y}\},\ M,\ S\,\rangle\Downarrow_{(h,s)}.$$

Proof. By induction over the length of the computation. □

With these technical lemmas at hand we can proceed with the proof of *let-let*. The law is symmetric so it suffices to prove one direction.

Proof. Assume the side conditions i.e., that

$$\operatorname{dom}\Gamma \cup \operatorname{dom}\Delta \subseteq \mathsf{FV}(M)$$

and

$$|\Gamma| = |\Delta|.$$

Because of the standard free-variable convention [Bar81] we know that all bound variables in in any instance of the law are distinct, and that they are disjoint from the free variables. We will use this silently throughout the proof. A first consequence is that

$$\mathsf{FV}(\mathsf{let}\ \Gamma\{{}^{v}_{w}x = \mathsf{let}\ \Delta\ \mathsf{in}\ M\}\ \mathsf{in}\ N) = \mathsf{FV}(\mathsf{let}\ \Delta\{{}^{v}_{w}x = \mathsf{let}\ \Gamma\ \mathsf{in}\ M\}\ \mathsf{in}\ N).$$

By the context lemma it is enough to show that for all Γ_0, S_0 and σ,

$$\begin{array}{l}\langle\, \Gamma_0,\ (\mathsf{let}\ \Gamma\{{}^{v}_{w}x = \mathsf{let}\ \Delta\ \mathsf{in}\ M\}\ \mathsf{in}\ N)\sigma,\ S_0\,\rangle\Downarrow_{(h,s)} \\ \Longrightarrow \\ \langle\, \Gamma_0,\ (\mathsf{let}\ \Delta\{{}^{v}_{w}x = \mathsf{let}\ \Gamma\ \mathsf{in}\ M\}\ \mathsf{in}\ N)\sigma,\ S_0\,\rangle\Downarrow_{(h,s)}.\end{array}$$

Thus assume

$$\langle\, \Gamma_0,\ (\mathsf{let}\ \Gamma\{{}^{v}_{w}x = \mathsf{let}\ \Delta\ \mathsf{in}\ M\}\ \mathsf{in}\ N)\sigma,\ S_0\,\rangle\Downarrow_{(h,s)}$$

i.e., that

$$\begin{array}{ll} & \langle\, \Gamma_0,\ (\mathsf{let}\ \Gamma\{{}^{v}_{w}x = \mathsf{let}\ \Delta\ \mathsf{in}\ M\}\ \mathsf{in}\ N)\sigma,\ S_0\,\rangle \\ \to_{(h,s)} & \langle\, \Gamma_0\Gamma\sigma\{{}^{v}_{w}x = (\mathsf{let}\ \Delta\ \mathsf{in}\ M)\sigma\},\ N\sigma,\ S_0\,\rangle \\ \gtrsim & \langle\, \Gamma_1\Gamma\sigma\{{}^{v}_{w}x = (\mathsf{let}\ \Delta\ \mathsf{in}\ M)\sigma\},\ N\sigma,\ S_1\,\rangle\Downarrow_{(h,s)}\end{array}$$

We have assumed, without loss of generality, that σ acts as the identity on the variables bound by the lets. Let $\vec{x}$ be a vector with the free variables of $(\mathsf{let}\ \Delta\ \mathsf{in}\ M)\sigma$. Then $\Lambda\vec{x}.(\mathsf{let}\ \Delta\ \mathsf{in}\ M)\sigma$ is a closed linear plug and

$$\begin{array}{ll} & \langle\, \Gamma_1\Gamma\sigma\{{}^{v}_{w}x = \xi\,\vec{x}\},\ N\sigma,\ S_1\,\rangle[{}^{\Lambda\vec{x}.(\mathsf{let}\ \Delta\ \mathsf{in}\ M)\sigma}/_{\xi}] \\ \equiv & \langle\, \Gamma_1\Gamma\sigma\{{}^{v}_{w}x = (\mathsf{let}\ \Delta\ \mathsf{in}\ M)\sigma\},\ N\sigma,\ S_1\,\rangle\Downarrow_{(h,s)}\end{array}$$

so it follows by the Uniform Computation Lemma for heap holes that either

$$\langle\, \Gamma_1\Gamma\sigma\{{}^{v}_{w}x = \xi\,\vec{x}\},\ N\sigma,\ S_1\,\rangle\Downarrow_{(h,s)}$$

or

$$\langle\, \Gamma_1\Gamma\sigma\{{}^{v}_{w}x = \xi\,\vec{x}\},\ N\sigma,\ S_1\,\rangle \to^{n}_{(h,s)} \langle\, \Delta_0,\ \xi\,\vec{x},\ T_0\,\rangle$$

and

$$\langle\, \Delta_0,\ \xi\,\vec{x},\ T_0\,\rangle[{}^{\Lambda\vec{x}.(\mathsf{let}\ \Delta\ \mathsf{in}\ M)\sigma}/_{\xi}]\Downarrow_{(h,s)}.$$

Before we consider the two different cases let us make the following remarks. From the first side condition of the law we know that $\mathrm{dom}\,\Gamma \subseteq \mathsf{FV}(M)$ and since $\mathrm{dom}\,\Gamma \cap \mathrm{dom}\,\Delta = \emptyset$ (the free variable convention) we have that

$$\mathrm{dom}\,\Gamma\sigma \subseteq \mathsf{FV}((\mathsf{let}\ \Delta\ \mathsf{in}\ M)\sigma) = \{\vec{x}\}.$$

Also from the free variable convention

$$\operatorname{dom}\Gamma\sigma \cap \mathsf{FV}(\Gamma_1, M\sigma, S_1) = \emptyset.$$

Let $\vec{y}$ be the free variables of $(\mathsf{let}\ \Gamma\ \mathsf{in}\ M)\sigma$. Then we have that

$$\begin{array}{ll} & (\mathsf{FV}(\Gamma\sigma) \cup \{\vec{x}\}) \setminus \operatorname{dom}\Gamma\sigma \\ = & (\mathsf{FV}(\Gamma\sigma) \cup \mathsf{FV}((\mathsf{let}\ \Delta\ \mathsf{in}\ M)\sigma)) \setminus \operatorname{dom}\Gamma\sigma \\ = & (\mathsf{FV}(\Delta\sigma) \cup \mathsf{FV}((\mathsf{let}\ \Gamma\ \mathsf{in}\ M)\sigma)) \setminus \operatorname{dom}\Delta\sigma \\ = & (\mathsf{FV}(\Delta\sigma) \cup \{\vec{y}\}) \setminus \operatorname{dom}\Delta\sigma \end{array}$$

where we have used that the variables bound by the lets are distinct and disjoint from the free variables. Now consider the first case. The second side condition of the law specifies that $|\Gamma| = |\Delta|$, i.e., that Γ and Δ takes up the same amount of space. Thus from Lemma 8.9 and our previous remarks it follows that

$$\langle\, \Gamma_1\Delta\sigma\{{}^{v}_{w}x = \xi'\,\vec{y}\},\ N\sigma,\ S_1\,\rangle\Downarrow_{(h,s)}.$$

By filling the hole with $\Lambda\vec{y}.(\mathsf{let}\ \Gamma\ \mathsf{in}\ M)\sigma$ it follows that

$$\langle\, \Gamma_1\Delta\sigma\{{}^{v}_{w}x = (\mathsf{let}\ \Gamma\ \mathsf{in}\ M)\sigma\},\ N\sigma,\ S_1\,\rangle\Downarrow_{(h,s)}.$$

so we have

$$\begin{array}{ll} & \langle\, \Gamma_0,\ (\mathsf{let}\ \Delta\{{}^{v}_{w}x = \mathsf{let}\ \Gamma\ \mathsf{in}\ M\}\ \mathsf{in}\ N)\sigma,\ S_0\,\rangle \\ \to & \langle\, \Gamma_0\Delta\sigma\{{}^{v}_{w}x = (\mathsf{let}\ \Gamma\ \mathsf{in}\ M)\sigma\},\ N\sigma,\ S_0\,\rangle \\ \gg & \langle\, \Gamma_1\Delta\sigma\{{}^{v}_{w}x = (\mathsf{let}\ \Gamma\ \mathsf{in}\ M)\sigma\},\ N\sigma,\ S_1\,\rangle\Downarrow_{(h,s)}. \end{array}$$

It only remains to note that

$$\begin{array}{l} |\langle\, \Gamma_0,\ (\mathsf{let}\ \Delta\{{}^{v}_{w}x = \mathsf{let}\ \Gamma\ \mathsf{in}\ M\}\ \mathsf{in}\ N)\sigma,\ S_0\,\rangle| \\ \qquad = |\langle\, \Gamma_0,\ (\mathsf{let}\ \Gamma\{{}^{v}_{w}x = \mathsf{let}\ \Delta\ \mathsf{in}\ M\}\ \mathsf{in}\ N)\sigma,\ S_0\,\rangle| \leq (h,s). \end{array}$$

Now consider the second case. From our previous remarks and

$$\langle\, \Gamma_1\Gamma\sigma\{{}^{v}_{w}x = \xi\,\vec{x}\},\ N\sigma,\ S_1\,\rangle \twoheadrightarrow^{n}_{(h,s)} \langle\, \Delta_0,\ \xi\,\vec{x},\ T_0\,\rangle$$

we know that $\Gamma\sigma$ must be a part of Δ_0, i.e., by Lemma 8.7,

$$\langle\, \Delta_0,\ \xi\,\vec{x},\ T_0\,\rangle \equiv \langle\, \Gamma_2\Gamma\sigma,\ \xi\,\vec{x},\ S_2\,\rangle$$

for some Γ_2 and S_2. Thus we have

$$\langle\, \Gamma_1\Gamma\sigma\{{}^{v}_{w}x = \xi\,\vec{x}\},\ N\sigma,\ S_1\,\rangle \twoheadrightarrow^{n}_{(h,s)} \langle\, \Gamma_2\Gamma\sigma,\ \xi\,\vec{x},\ S_2\,\rangle$$

and

$$\begin{array}{ll} & \langle\, \Delta_0,\ \xi\,\vec{x},\ T_0\,\rangle[\Lambda\vec{x}.(\mathsf{let}\ \Delta\ \mathsf{in}\ M)\sigma/\xi] \\ \equiv & \langle\, \Gamma_2\Gamma\sigma,\ \xi\,\vec{x},\ S_2\,\rangle[\Lambda\vec{x}.(\mathsf{let}\ \Delta\ \mathsf{in}\ M)\sigma/\xi] \\ \equiv & \langle\, \Gamma_2\Gamma\sigma,\ (\mathsf{let}\ \Delta\ \mathsf{in}\ M)\sigma,\ S_2\,\rangle\Downarrow_{(h,s)}. \end{array}$$

From the latter it follows that

$$\begin{array}{ll} & \langle\, \Gamma_2\Gamma\sigma,\ (\mathsf{let}\ \Delta\ \mathsf{in}\ M)\sigma,\ S_2\,\rangle \\ \to & \langle\, \Gamma_2\Gamma\sigma\Delta\sigma,\ M\sigma,\ S_2\,\rangle \\ \gtrdot & \langle\, \Gamma_3\Gamma\sigma\Delta\sigma,\ M\sigma,\ S_3\,\rangle\Downarrow_{(h,s)}. \end{array}$$

Apart from the step marked with $(*)$ it follows, by the definition of transitions and garbage collection using

$$\mathsf{FV}(\mathsf{let}\ \Gamma\{^v_w x = \mathsf{let}\ \Delta\ \mathsf{in}\ M\}\ \mathsf{in}\ N) = \mathsf{FV}(\mathsf{let}\ \Delta\{^v_w x = \mathsf{let}\ \Gamma\ \mathsf{in}\ M\}\ \mathsf{in}\ N),$$

that

$$\begin{array}{ll} & \langle\, \Gamma_0,\ (\mathsf{let}\ \Delta\{^v_w x = \mathsf{let}\ \Gamma\ \mathsf{in}\ M\}\ \mathsf{in}\ N)\sigma,\ S_0\,\rangle \\ \to & \langle\, \Gamma_0\Delta\sigma\{^v_w x = (\mathsf{let}\ \Gamma\ \mathsf{in}\ M)\sigma\},\ N\sigma,\ S_0\,\rangle \\ \gtrdot & \langle\, \Gamma_1\Delta\sigma\{^v_w x = (\mathsf{let}\ \Gamma\ \mathsf{in}\ M)\sigma\},\ N\sigma,\ S_1\,\rangle \\ \twoheadrightarrow^n_{(h,s)} & \langle\, \Gamma_2\Delta\sigma,\ (\mathsf{let}\ \Gamma\ \mathsf{in}\ M)\sigma,\ S_2\,\rangle \qquad (*) \\ \to & \langle\, \Gamma_2\Delta\sigma\Gamma\sigma,\ M\sigma,\ S_2\,\rangle \\ \gtrdot & \langle\, \Gamma_3\Delta\sigma\Gamma\sigma,\ M\sigma,\ S_3\,\rangle\Downarrow_{(h,s)}. \end{array}$$

The step marked with $(*)$ is a bit involved. Recall that we showed that

$$\langle\, \Gamma_1\Gamma\sigma\{^v_w x = \xi\,\vec{x}\},\ N\sigma,\ S_1\,\rangle \twoheadrightarrow^n_{(h,s)} \langle\, \Gamma_2\Gamma\sigma,\ \xi\,\vec{x},\ S_2\,\rangle.$$

The second side condition of the law specifies that $|\Gamma| = |\Delta|$, i.e., that Γ and Δ takes up the same amount of space. Thus from Lemma 8.8 and our previous remarks it follows that

$$\langle\, \Gamma_1\Delta\sigma\{^v_w x = \xi\,\vec{x}\},\ N\sigma,\ S_1\,\rangle \twoheadrightarrow^n_{(h,s)} \langle\, \Gamma_2\Delta\sigma,\ \xi'\,\vec{y},\ S_2\,\rangle.$$

Now $(*)$ follows by plugging the hole with $\Lambda\vec{y}.(\mathsf{let}\ \Gamma\ \mathsf{in}\ M)\sigma$. It remains to show that the configurations in the transition sequence uses at most (h, s) space. For the first configuration we have

$$\begin{array}{l} |\langle\, \Gamma_0,\ (\mathsf{let}\ \Delta\{^v_w x = \mathsf{let}\ \Gamma\ \mathsf{in}\ M\}\ \mathsf{in}\ N)\sigma,\ S_0\,\rangle| \\ \qquad\qquad = |\langle\, \Gamma_0,\ (\mathsf{let}\ \Gamma\{^v_w x = \mathsf{let}\ \Delta\ \mathsf{in}\ M\}\ \mathsf{in}\ N)\sigma,\ S_0\,\rangle| \le (h, s), \end{array}$$

and for the second configuration

$$|\langle\, \Gamma_2\Delta\sigma,\ (\mathsf{let}\ \Gamma\ \mathsf{in}\ M)\sigma,\ S_2\,\rangle| = |\langle\, \Gamma_2\Gamma\sigma,\ (\mathsf{let}\ \Delta\ \mathsf{in}\ M)\sigma,\ S_2\,\rangle| \le (h, s).$$

Thus

$$\langle\, \Gamma_0,\ (\mathsf{let}\ \Delta\{^v_w x = \mathsf{let}\ \Gamma\ \mathsf{in}\ M\}\ \mathsf{in}\ N)\sigma,\ S_0\,\rangle\Downarrow_{(h,s)}$$

as required. □

9 Related Work

Improvement theory was first developed in the call-by-name setting [San95, San91, San96] for the purpose of reasoning about running-times of programs. Moran and Sands [MS99a] developed a call-by-need time-improvement theory, together with a variety of induction principles. This present work, and its predecessors [GS99, GS01] is the only attempt (of which we are aware) which formalises space safety properties of local (non-whole-program) transformations.

Other related work includes the development of "space-aware" operational models for call-by-need languages [Ses97, Ros96, BLR96, BR00b], studies of space-safety properties of global transformations [Min99, Min00] and of the relative efficiency of different abstract machines [BG96, Cli98, BR00a, Min00]. Morrisett and Harper [MH98] use a similar style of abstract machine description to that used here in order to investigate the semantics of memory management in an ML-like language (see also [MFH95]). They give abstract specifications of garbage collection, and prove the correctness of a particular type-based collection scheme.

Minamide [Min00] suggests an alternative to our definition of improvement based on additive constant factors. Its properties are not studied for any particular language, although we suspect that it would fail to satisfy the syntactic continuity property, so would not serve as an alternative to strong improvement.

A number of insights into space problems of lazy evaluation – which we have found useful – can be found in a range of sources, e.g., [Jon92, Wad87, Spa93, PJ87, Hug83, RW93, Röj95].

10 Conclusions and Future Work

We have presented a surprisingly[7] rich operational theory for the space use of call-by-need programs, based on a space improvement ordering on programs. The theory allows one to argue that transforming a program fragment M into N is space safe in the sense that replacing M by N in any program can never lead to asymptotically worse space (heap or stack) behaviour. We also showed that the asymptotic space improvement relation is semantically badly behaved, but that the theory of strong space improvement possesses a fixed-point induction theorem which permits the derivation of improvement properties for recursive definitions. With the help of this tool we explored the landscape of space improvement by considering a range of classical program transformations.

Areas for further work include the introduction of context information to the theory in order to represent constraints on the whole-program context which can be used to help establish space improvements. Another interesting direction for future work would be to consider the space safety of a larger-scale program transformation, such as deforestation [Wad90].

[7] At least, suprising to us!

References

[Aug87] Lennart Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Department of Computer Science, Chalmers University of Technology, 1987.

[Bar81] H. Barendregt. *The Lambda Calculus.* North Holland, 1981.

[BG96] Guy E. Blelloch and John Greiner. A provably time and space efficient implementation of NESL. In *Proc. ICFP'96*, pages 213–225, 1996.

[BLR96] Z.-E.-A. Benaissa, P. Lescanne, and K. H. Rose. Modeling sharing and recursion for weak reduction strategies using explicit substitution. In *Proc. PLILP'96*, volume 1140 of *LNCS*, pages 393–407. Springer-Verlag, 1996.

[BR00a] Adam Bakewell and Colin Runciman. A model for comparing the space usage of lazy evaluators. In *Proceedings of Principles and Practice of Declarative Programming*, September 2000.

[BR00b] Adam Bakewell and Colin Runciman. A space semantics for core haskell. In *Proceedings of the Haskell Workshop*, September 2000.

[Chi93] Wei-Ngan Chin. Towards an automated tupling strategy. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Copenhagen, Denmark, 1993.

[Cli98] William D. Clinger. Proper tail recursion and space efficiency. In *Proc. PLDI'98*, 1998.

[GP98] A. D. Gordon and A. M. Pitts, editors. *Higher Order Operational Techniques in Semantics.* Cambridge University Press, 1998.

[GS99] J. Gustavsson and D. Sands. A foundation for space-safe transformations of call-by-need programs. In A. D. Gordon and A. M.Pitts, editors, *The Third International Workshop on Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 1999.

[GS01] J. Gustavsson and D. Sands. Possibilities and limitations of call-by-need space improvement. In *Proceedings of the International Conference on Functional Programming*, September 2001. To Appear.

[Hug83] R. J. M. Hughes. *The Design and Implementation of Programming Languages.* PhD thesis, Programming Research Group, Oxford University, July 1983.

[Jon92] Richard Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. POPL'93*, pages 144–154. ACM Press, January 1993.

[MFH95] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract Models of Memory Management. In *Proc. FPCA'95*, pages 66–77. ACM Press, June 1995.

[MH98] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In Gordon and Pitts [GP98], pages 175–226.

[Mil77] R. Milner. Fully abstract models of the typed λ-calculus. *Theoretical Computer Science*, 4:1–22, 1977.

[Min99] Yasuhiko Minamide. Space-profiling semantics of the call-by-value lambda calculus and the cps transformation. In *Proc. HOOTS III*, 1999. To appear as ENTCS.

[Min00] Yasuhiko Minamide. A new criterion for safe program transformations. 2000. To appear as ENTCS.

[MS99a] Andrew Moran and David Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Proc. POPL'99*, pages 43–56. ACM Press, January 1999.

[MS99b] Andrew Moran and David Sands. Improvement in a lazy context: An operational theory for call-by-need (extended version). Extended version of [MS99a], 1999.

[MST96] Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. From operational semantics to domain theory. *Information and Computation*, 128(1):26–47, 10 July 1996.

[Pet77] A. Pettorossi. Transformation of programs and use of tupling strategy. In *Proceedings Informatica 77, Bled, Yugoslavia.*, pages 1–6, 1977.

[Pit94] A. M. Pitts. Some notes on inductive and co-inductive techniques in the semantics of functional programs. Notes Series BRICS-NS-94-5, BRICS, Department of Computer Science, University of Aarhus, December 1994.

[PJ87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[PJ92] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2), April 1992.

[PJHA+99] Simon Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98: A non-strict, purely functional language, February 1999. Available at www.haskell.org.

[PJL91] Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens. In *Proocedings of the Conference on Functional Programming and Computer Architecture*, September 1991.

[PJPS96] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proc. ICFP'96*, pages 1–12. ACM Press, May 1996.

[PJS98] S. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998.

[Röj95] Niklas Röjemo. *Garbage collection, and memory efficiency, in lazy functional languages.* PhD thesis, Chalmers Tekniska Högskola, 1995.

[Ros96] K. H. Rose. *Operational Reduction Models for Functional Programming Languages.* PhD thesis, DIKU, University of Copenhagen, Denmark, February 1996. available as DIKU report 96/1.

[RR96] Colin Runciman and Niklas Röjemo. Lag, Drag, Void and Use – Heap Profiling and Space-efficient Compilation Revisited. In *Proceedings of 1st Intl. Conf. on Functional Programming (ICFP'96)*, pages 34–41. ACM Press, May 1996.

[RW93] Colin Runciman and David Wakeling. Heap Profiling of Lazy Functional Programs. *Journal of Functional Programming*, 3(2):217–245, April 1993.

[San91] D. Sands. Operational theories of improvement in functional languages (extended abstract). In *Proc. 1991 Glasgow Functional Programming Workshop*, Workshops in Computing Series, pages 298–311. Springer-Verlag, August 1991.

[San95] D. Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995.

[San96] D. Sands. Total correctness by local improvement in the transformation of functional program. *ACM TOPLAS*, 18(2):175–234, March 1996.

[San98] D. Sands. Computing with contexts: A simple approach. In A. D. Gordon, A. M. Pitts, and C. L. Talcott, editors, *Proc. HOOTS II*, volume 10 of *ENTCS*. Elsevier Science Publishers B.V., 1998.

[Ses97] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[Smi92] Scott F. Smith. From operational to denotational semantics. In Steven Brookes *et al.*, editor, *7th International Conference on Mathematical Foundations of Programming Semantics, Pittsburgh PA*, pages 54–76, Berlin, 1992. Springer Verlag. Lecture Notes in Computer Science Volume 598.

[Spa93] Jan Sparud. Fixing Some Space Leaks without a Garbage Collector. In *Proc. 6th Int'l Conf. on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 117–122. ACM Press, June 1993.

[Wad87] P. Wadler. Fixing Some Space Leaks with a Garbage Collector. *Software Practice and Experience*, September 1987.

[Wad90] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

A Language Extension: Boxing and Unboxing

In this section we extend our language with unboxed integers. The extension is not essential in the same way as pattern bindings because (we believe) that it does not add to the intensional expressiveness of the language. But we found explicit constructs for boxing and unboxing to be very useful in calculations of strong improvement because is allows the proof and the required basic laws to be more fine-grained. The usefulness of these language constructs when performing program transformation is also noted by Peyton Jones and Launchbury [PJL91].

We will use $M\#$ to range over unboxed expressions. Following Peyton Jones and Launchbury we will use a # to distinguish unboxed constants and variables and operations from their boxed counterparts. The grammar of unboxed terms is as follows.

$$\text{Unboxed Terms} \quad M\# \quad ::= \quad n\# \mid i\# \mid M\#_0 +\# M\#_1$$

We extend our term language with a construct $\mathsf{Int}\, M\#$, for boxing an unboxed integer, where Int can be thought of as a kind of constructor. Following Peyton Jones and Launchbury we use the case expression syntax for unboxing:

$\mathsf{case}\ M\ \mathsf{of}\ \{\mathsf{Int}\, i\# \to N\}$. Finally we have a construct for comparing unboxed values which yields a boolean: $M\#_0 \leq\# M\#_1$. With these constructs we can encode boxed addition like

$$M + N \stackrel{\text{def}}{=} \mathsf{case}\ M\ \mathsf{of}\ \{\mathsf{Int}\, i\# \to \mathsf{case}\ N\ \mathsf{of}\ \{\mathsf{Int}\, j\# \to \mathsf{Int}\, (i\# +\# j\#)\}\}$$

Next, we extend our abstract machine with boxing and unboxing. Reduction contexts now also includes $\mathsf{case}\ [\cdot]\ \mathsf{of}\ \{\mathsf{Int}\, i\# \to N\}$ and we extend $\rightsquigarrow$ with the following clause

$$\mathsf{case}\ \mathsf{Int}\, n\#\ \mathsf{of}\ \{\mathsf{Int}\, i\# \to M\} \rightsquigarrow M[{}^{n\#}/_{i\#}]$$

where $[{}^{n\#}/_{i\#}]$ is a substitution of an unboxed term for and unboxed variable. For a closed unboxed term $M\#$ we will write $[\![M\#]\!]$ for the result of evaluating $M\#$. We will not account for any stack space that may be needed when evaluating an unboxed term. Our semantics is still within a program size dependent constant factor because the amount of stack that is needed in an implementation is bounded by the size of the unboxed terms in the program. The additional abstract machine rules that we need are given below.

$$\langle \Gamma,\ \mathsf{Int}\, M\#,\ S \rangle \to \langle \Gamma,\ \mathsf{Int}\, n\#,\ S \rangle \quad \text{if } [\![M\#]\!] = n \qquad (\mathit{Box})$$

$$\langle \Gamma,\ M\#_0 \leq\# M\#_1,\ S \rangle \to \begin{cases} \langle \Gamma,\ \mathsf{true},\ S \rangle & \text{if } [\![M\#_0]\!] \leq [\![M\#_1]\!] \\ \langle \Gamma,\ \mathsf{false},\ S \rangle & \text{otherwise} \end{cases} \qquad (\mathit{Compare})$$

Note that the unboxing operation is modelled by the rule schemas *Push* and *Reduce*:

$$\langle \Gamma,\ R[M],\ S \rangle \to \langle \Gamma,\ M,\ R : S \rangle \qquad (\mathit{Push})$$

$$\langle \Gamma,\ V,\ R : S \rangle \to \langle \Gamma,\ M,\ S \rangle \quad \text{if } R[V] \rightsquigarrow M \qquad (\mathit{Reduce})$$

In Figure 11 we have collected some laws for unboxed terms and in Figure 12 some laws for boxing and unboxing. We used these laws when we proved Proposition 6.8.

B Language Extension: Pattern Bindings

In this appendix we extend our language with pattern bindings.

Bindings in let expressions may now take the following form $c\,\vec{x} = M$. The heap may also contain the new form of binding but also *indirections* which is of the form $x \mapsto y$ and can be thought of as a binding $x = y$ which is treated specially by the garbage collector. We also need a new form of stack element which we call *pattern binding matchers*. They take the form $\#c\,\vec{x}, \Pi_{c_i}$, indicating

$$\mathsf{Int}\,(M\#\mathbin{+\#}0\#) \mathrel{\lessapprox\gtrapprox} \mathsf{Int}\,M\#$$
$$\mathsf{Int}\,(0\#\mathbin{+\#}M\#) \mathrel{\lessapprox\gtrapprox} \mathsf{Int}\,M\#$$
$$\mathsf{Int}\,((M\#_0\mathbin{+\#}M\#_1)\mathbin{+\#}M\#_2) \mathrel{\lessapprox\gtrapprox} \mathsf{Int}\,(M\#_0\mathbin{+\#}(M\#_1\mathbin{+\#}M\#_2))$$
$$\mathsf{Int}\,(M\#_0\mathbin{+\#}M\#_1) \mathrel{\lessapprox\gtrapprox} \mathsf{Int}\,(M\#_1\mathbin{+\#}M\#_0)$$

Figure 11: Laws for unboxed terms.

$$^{w}\mathsf{case}\ \mathsf{Int}\,M\#\ \mathsf{of}\ \{\mathsf{Int}\,i\#\to N\}$$
$$\mathrel{\lessapprox\gtrapprox} {}^{w\curlyvee}N[M\#/i\#]$$

$$\mathsf{let}\ \Gamma\{{}^{v}_{w}x=\mathsf{Int}\,M\#\}\ \mathsf{in}\ \mathbb{C}[x]$$
$$\mathrel{\gtrapprox} \mathsf{let}\ \Gamma\{{}^{v}_{w}x=\mathsf{Int}\,M\#\}\ \mathsf{in}\ \mathbb{C}[{}^{\{x\}}\mathsf{Int}\,M\#]$$

$$\mathsf{let}\ \Gamma\{{}^{v}_{w}x=\mathsf{Int}\,M\#\}\ \mathsf{in}\ \mathbb{C}[x]$$
$$\mathrel{\lessapprox} \mathsf{let}\ \Gamma\{{}^{v}_{w}x=\mathsf{Int}\,M\#\}\ \mathsf{in}\ \mathbb{C}[{}^{\{x\}v\curlyvee}\mathsf{Int}\,M\#]$$

$$\mathsf{let}\ \Gamma\{x=\mathsf{Int}\,M\#,\ y=\mathsf{Int}\,M\#\}\ \mathsf{in}\ M$$
$$\mathrel{\gtrapprox} \mathsf{let}\ \Gamma[x/y]\{x=\mathsf{Int}\,M\#\}\ \mathsf{in}\ M[x/y]$$

$$\mathsf{let}\ \Gamma\{x=\mathsf{Int}\,M\#,\ y=\mathsf{Int}\,M\#\}\ \mathsf{in}\ M$$
$$\mathrel{\lessapprox} \mathsf{let}\ \Gamma[x/y]\{{}_{2}x=\mathsf{Int}\,M\#\}\ \mathsf{in}\ M[x/y]$$

$$\mathsf{let}\ \{{}^{w_0}_{v}x=\Omega\}\ \mathsf{in}\ {}^{w_1+w_0}\mathsf{case}\ M\ \mathsf{of}\ \{\mathsf{Int}\,i\#\to N\}$$
$$\mathrel{\lessapprox\gtrapprox} \mathsf{let}\ \{{}^{w_0}_{v}x=M\}\ \mathsf{in}\ {}^{w_1}\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{Int}\,i\#\to N\}$$
if x is a dummy reference in N
but does not occur elsewhere in M or N.

$$\mathsf{let}\ \{{}^{v}_{w}x=M\}\ \mathsf{in}\ \mathbb{C}[\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{Int}\,i\#\to\mathbb{D}[x]\}]$$
$$\mathrel{\gtrapprox} \mathsf{let}\ \{{}^{v}_{w}x=M\}\ \mathsf{in}\ \mathbb{C}[\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{Int}\,i\#\to\mathbb{D}[{}^{\{x\}}\mathsf{Int}\,i\#]\}]$$

$$\mathsf{let}\ \{{}^{v}_{w}x=M\}\ \mathsf{in}\ \mathbb{C}[\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{Int}\,i\#\to\mathbb{D}[x]\}]$$
$$\mathrel{\lessapprox} \mathsf{let}\ \{{}^{v}_{w}x=M\}\ \mathsf{in}\ \mathbb{C}[\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{Int}\,i\#\to\mathbb{D}[{}^{\{x\}v\curlyvee}\mathsf{Int}\,i\#]\}]$$

Figure 12: Laws for boxing and unboxing.

$$\langle\, \Gamma\{c\,\vec{x} = M\},\ x_i,\ S\,\rangle \to \langle\, \Gamma,\ M,\ \#c\,\vec{x}, \Pi_{c_i} : S\,\rangle \qquad (\textit{Lookup})$$

$$\langle\, \Gamma,\ c\,\vec{y},\ \#c\,\vec{x}, \Pi_{c_i} : S\,\rangle \to \langle\, \Gamma\{\vec{x} \mapsto \vec{y}\},\ y_i,\ S\,\rangle \qquad (\textit{Match})$$

$$\langle\, \Gamma\{x \mapsto y\},\ x,\ S\,\rangle \to \langle\, \Gamma\{x \mapsto y\},\ y,\ S\,\rangle \qquad (\textit{Indirect})$$

Figure 13: Rules for pattern bindings

that the result of the current computation should be matched against $c\,\vec{x}$. If the matching succeeds $\vec{x}$ should be indirected to the components of the matched result and the computation proceed with i'th component. The $\vec{x}$ should be thought of as binding occurrences of the variables analogously to the variable in an update marker.

The abstract machine rules for pattern bindings is given in figure 13. To evaluate a variable x_i bound in a pattern binding, we remove the binding $c\,\vec{x} = M$ from the heap and start evaluating M, with a pattern binding matcher $\#c\,\vec{x}, \Pi_{c_i}$ pushed onto the stack. Rule (*Match*) applies when this evaluation is finished, and match the result against the pattern in the projection. If the matching succeeds the indirections $\vec{x} \mapsto \vec{y}$ are added to the heap and the computation proceeds with the i'th component of the result. The rule (*Indirect*) specifies that indirections should be followed.

We measure the heap space occupied by a pattern binding by counting the number of variables bound by the pattern. A pattern binding matcher takes up one unit of stack and just as for update markers we count the variable bound by the matcher as occupying heap space.

Garbage collection in the presence of pattern bindings is a bit involved for two reasons. Firstly, because the update rule creates indirections in the heap and we will allow the garbage collector to shortcut them, and secondly, a pattern binder $c\,\vec{x} = M$ binds all the $\vec{x}$ at the same time but we want to garbage collect them individually. We model shortcutting of indirections in the garbage collector by the following rule.

$$\langle\, \Gamma\{x \mapsto y\},\ M,\ S\,\rangle \gg \langle\, \Gamma[y/x],\ M[y/x],\ S[y/x]\,\rangle \quad \text{if } x \neq y.$$

Note that we cannot shortcut an indirection which forms a cycle (the side condition $x \neq y$) since it could cause the configuration to become open. However if there is no other free occurrences of x in the configuration we may remove the indirection just as we can remove other bindings in the heap. To model the garbage collection of an individual variable x_i in a pattern binder $c\,\vec{x} = M$ we use a special placeholder: _. We may, if there is no free occurrence of x_i, replace x_i with _. Then, when we count the heap usage of the binder we don't count the _'s. If all the variables in a pattern binding is dead we may also remove the binding just as we remove ordinary bindings. Recall that we consider the variables in a pattern binding matcher on the stack as binding occurrences occupying

heap. Consequently we may replace individual variables in a pattern binding matcher with an _ in the same way as for a pattern binding. Note however that we may not remove the pattern binding matcher itself even if all the variables are dead since the pattern binding matcher also performs a projection.

Paper II

On Usage Analyses for Work and Space Safe Inlining

On Usage Analyses for Work and Space Safe Inlining

Jörgen Gustavsson David Sands

Abstract

To inline function calls can be a very worthwhile program transformation. But, as is well-known, in a call-by-need language the transformation risks duplicating computation, and this can lead to an asymptotically worse program – in both space and time. A number of researchers, e.g., Turner, Wadler and Mossin [TWM95], have sought to find criteria for when such transformations are work-safe, based on notions of "used at most once". Despite the fact that Turner et al discuss inlining of "used-once" bindings in some detail, as far as we are aware, it remains an open problem to actually prove that these criteria actually do guarantee work-safety. Another question (one which to our knowledge has not even been posed) is whether the "used at most once" criteria might also guarantee space safety.

In this paper we show that the "used at most once" criteria alone is not enough to guarantee space-safety. We therefore strengthen the use-once criteria and show that the stronger criteria is enough to guarantee both work and space safety. Some of the published usage analyses, including the analysis by Turner et al, satisfy the stronger criteria so work and space safety follows from our result. Some other analyses, e.g., [Mog97] do not satisfy the additional criteria, and we believe that as a result those analyses do not provide conditions for space-safe inlining.

1 Introduction

Most implementations of non-strict functional languages rely on a call-by-need evaluation. Call-by-need optimises call-by-name by ensuring that arguments to functions are evaluated only if needed and *at most once*. In our opinion, call-by-need goes beyond being an internal compiler optimisation because it affects the *asymptotic* time and space complexity of programs and the programmer must be able to trust that the call-by-need semantics is respected.

The state-of-the-art compilers for lazy languages are based on intensive program transformations – inspired by the clean equational theory of pure lazy languages [PJS98]. But a compiler that wants to respect the intentional call-by-need semantics cannot rely directly on the equational theory because equivalent

programs may have a different asymptotic complexity. Even β-reduction

$$(\lambda x.M)\, N \Rightarrow M[{}^{N}/_{x}],$$

the simplest of laws, can lead to an exponential blow up in time complexity. Why is this? Well, consider the function f below which computes 2^n in $\mathcal{O}(n)$ steps.

$$\begin{aligned} f\, 0 &= 1 \\ f\, n &= \mathit{double}\ (f\ (n-1)) \end{aligned}$$

Suppose also that the programmer has defined *double* as

$$\mathit{double}\ x = x + x.$$

He or she would probably had been better off with the definition

$$\mathit{double}\ x = 2 * x$$

but with call-by-need evaluation x is evaluated only once anyway so his/her definition is only a compiler dependent constant factor worse. Unless the compiler "optimises" the program by *inlining* the call to *double*, i.e., it replaces *double* by its definition and performs the β-reduction:

$$\begin{aligned} f\, 0 &= 1 \\ f\, n &= f\ (n-1) + f\ (n-1). \end{aligned}$$

The result of the transformation is a function which computes 2^n in $\mathcal{O}(2^n)$ steps. With the latter definition of *double* the transformation wouldn't have duplicated any computation. Intuitively, this is because the latter definition uses its argument *once* rather than twice.

A number of researchers have sought to find criteria for when β-reduction is *work-safe*, based on notions of "used at most once", e.g, Turner, Wadler and Mossin [TWM95]. In Section 5 we give an overview of this line of research. The more recent usage analyses, starting from Turner et al [TWM95] have been proved sound in the sense that when the analysis claims that an argument is used at most once then it is indeed the case. But despite the fact that Turner et al discuss inlining of used-once bindings in some detail, as far as we are aware, it remains an open problem to actually prove that any of the usage analyses guarantee work-safety of β-reduction . Another question (one which to our knowledge has not even been posed) is whether usage analyses might also guarantee *space safety*.

Rather than showing work and space safety for any particular analysis we pose the question: can the the intuitive semantic criteria "used at most once" guarantee work and space safety? One problem is that time and space safety

do not go hand in hand. Sometimes inlining can lead to asymptotically worse space behaviour *even when it is work-safe*. For example, let g be defined as

$$g\,x\,xs = x + traverse\,xs + (\lambda y.1)\,x$$

where *traverse* is a function which traverses its input list and returns 0. Note that g uses its argument, x, only once. But it retains a reference to x until after $traverse\,xs$ have been evaluated.[1] Suppose we inline the call to g:

$$\begin{array}{l}\mathsf{let}\ \{xs = count\,\mathbf{k}\} \Rightarrow \mathsf{let}\ \{xs = count\,\mathbf{k}\} \\ \mathsf{in}\ g\,(head\,xs)\,xs \qquad\quad \mathsf{in}\ head\,xs + traverse\,xs + (\lambda y.1)\,(head\,xs)\end{array}$$

where *count* is a function which produces the list of integers counting down from its argument to zero. The inlining above is work-safe but not space safe: the left hand side can run in constant space but the right hand side requires heap space proportional to $\mathbf{k}$. This example is enough to show that the "used at most once" criteria alone is not enough to guarantee space safety. In this paper we will strengthen the use-once criteria so that it is enough to guarantee both work and space safety. Intuitively, the stronger criteria is that an argument must be used at most once and when it is used there may be no other references to the closure holding the argument. We will refer to the stronger criteria as the "use-once-don't-drag" criteria. The usage analyses by Gustavsson [Gus98, Gus99] and Gustavsson and Svenningsson [GS00] have already been proven to satisfy the "use-once-don't-drag" criteria and we believe that the analyses by Turner et al [TWM95] and Wansbrough and Peyton-Jones [WPJ99, WPJ00] do so as well. However the analyses by Sestoft [Ses91], Marlow [Mar93] and Mogensen [Mog97] do not satisfy the additional criteria, and we believe that as a result their analyses do not provide conditions for space-safe inlining.

Overview The remainder of the article is organised as follows. **Section 2** gives the syntax and operational semantics of our language and defines what we mean by the time consumption and space-use of programs. We also make precise the notion of "use-once-don't-drag". **Section 3** defines two notions of work and space safety. **Section 4** state and prove work and space safety of inlining "use-once-don't-drag" bindings. **Section 5** describes related work. **Section 6** concludes and proposes future work.

2 Operational Semantics

In this section we give the syntax and call-by-need operational semantics of our language in terms of an abstract machine. We define what we mean by the time consumption and space-use of programs and we make precise the criteria needed for space safety: that an argument is used at most once and that when it is used

[1] Assuming + evaluates from left to right.

there is no other references to the closure holding the argument. To this end we extend the language with a "use-once-don't-drag" application, $M \bullet N$ equipped with a direct operational interpretation.

Our language is an untyped lambda calculus with recursive lets, structured data, case expressions, bounded integers (ranged over by n and m) with addition and a zero test. We work with a restricted syntax in which arguments to functions (including constructors) are always variables, so applications take the form $M\,x$ rather than $M\,N$. The syntactic restriction is now rather standard, following its use in core language of the Glasgow Haskell compiler, *e.g.*, [PJPS96, PJS98], and in [Lau93, Ses97]. An unrestricted application $M\,N$ where N is not a variable will now be taken as syntactic sugar for $\mathsf{let}\ \{x = N\}\ \mathsf{in}\ M\,x$ where x is a fresh variable. And a "use-once-don't-drag" application, $M \bullet N$ is syntactic sugar for $\mathsf{let}\ \{x \stackrel{\bullet}{=} N\}\ \mathsf{in}\ M\,x$ where $x \stackrel{\bullet}{=} N$ is a "use-once-don't-drag" binding. Thus, the grammar of our language is as follows.

$$\begin{array}{lll}
\text{Terms} & L, M, N ::= & x \mid \lambda x.M \mid M\,x \mid c\,\vec{x} \\
& & \mid\ n \mid M + N \mid \mathsf{add}_n M \mid \mathsf{iszero}\,M \\
& & \mid\ \mathsf{let}\ \{\vec{B}\}\ \mathsf{in}\ N \mid \mathsf{case}\ M\ \mathsf{of}\ \{c_i\,\vec{x}_i \to N_i\} \\
\text{Bindings} & B ::= & x = M \mid x \stackrel{\bullet}{=} M
\end{array}$$

All constructors have a fixed arity, and are assumed to be saturated. By $c\,\vec{x}$ we mean $c\,x_1 \cdots x_n$. The only values are lambda expressions and fully-applied constructors. Throughout, x, y, z etc., will range over variables, c over constructor names, and V and W over values ($\lambda x.M \mid c\,\vec{x} \mid n$). We will write

$$\mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N$$

as a shorthand for $\mathsf{let}\ \{x_1 = M_1, \ldots, x_n = M_n\}\ \mathsf{in}\ N$ where the $\vec{x}$ are distinct, the order of bindings is not syntactically significant, and the $\vec{x}$ are considered bound in N *and* the $\vec{M}$ (so our lets are recursive). Similarly we write

$$\mathsf{case}\ M\ \mathsf{of}\ \{c_i\,\vec{x}_i \to N_i\}$$

for

$$\mathsf{case}\ M\ \mathsf{of}\ \{c_1\,\vec{x}_1 \to N_1 \mid \cdots \mid c_m\,\vec{x}_m \to N_m\}$$

where each $\vec{x}_i$ is a vector of distinct variables, and the c_i are distinct constructors. In addition, we will sometimes write *alts* as an abbreviation for case alternatives $\{c_i\,\vec{x}_i \to N_i\}$.

Our integers are bounded (i.e., for an integer n, $MININT \leq n \leq MAXINT$) so that they can be represented in constant space. For simplicity, no exception occurs at overflow. Instead the result wraps as in e.g., C. The functions add_n are included for convenience in the definition of the abstract machine, and represent an intermediate step in the addition of n to a term.

The free variables of a term M will be denoted $\mathsf{FV}(M)$; for a vector of terms $\vec{M}$, we will write $\mathsf{FV}(\vec{M})$. The only kind of substitution that we consider is *variable for variable*, with σ ranging over such substitutions. The simultaneous substitution of one vector of variables for another will be written $M[\vec{y}/\vec{x}]$, where the $\vec{x}$ are assumed to be distinct (but the $\vec{y}$ need not be).

2.1 The Abstract Machine

The semantics presented in this section is essentially Sestoft's "mark 1" abstract machine for laziness [Ses97].

Configurations Transitions are over configurations consisting of a *heap*, containing bindings, the expression currently being evaluated, and a *stack*. We write $\langle\, \Gamma,\ M,\ S \,\rangle$ for the abstract machine configuration with heap Γ, expression M, and stack S and we will use Σ and Φ to range over such configurations. A heap is a set of bindings; we denote the empty heap by $\emptyset$, and the addition of a group of fresh bindings $\vec{B}$ to a heap Γ by juxtaposition: $\Gamma\{\vec{B}\}$. The stack written $a : S$ will denote the stack S with a pushed on the top. The empty stack is denoted by ϵ.

Stack elements are either:

- a *reduction context*, or
- an *update marker* $\#x$, indicating that the result of the current computation should be bound to the variable x in the heap.

The reduction contexts on the stack are shallow contexts containing a single hole in a "reduction" position - i.e. in a position where the current computation is being performed. They are defined as:

$$R ::= [\cdot]\, x \mid \mathsf{case}\ [\cdot]\ \mathsf{of}\ \{c_i\, \vec{x}_i \to N_i\} \mid [\cdot] + M \mid \mathsf{add}_n[\cdot] \mid \mathsf{iszero}\,[\cdot]$$

We will refer to the set of variables bound by Γ as $\mathrm{dom}\,\Gamma$, and to the set of variables marked for update in a stack S as $\mathrm{dom}\,S$. Update markers should be thought of as binding occurrences of variables. A configuration is *well-formed* if $\mathrm{dom}\,\Gamma$ and $\mathrm{dom}\,S$ are disjoint. We write $\mathrm{dom}(\Gamma, S)$ for their union. For a configuration $\langle\, \Gamma,\ M,\ S \,\rangle$ to be closed, any free variables in Γ, M, and S must be contained in $\mathrm{dom}(\Gamma, S)$.

Garbage collection We cannot reason about space usage without modelling garbage collection. During a computation, garbage collection allows us to decrease the amount of space used by a configuration. It is modelled simply by the removal of any number of bindings and update markers from the heap and the stack respectively, *providing that the configuration remains closed.*

$$\langle\, \Gamma\{x = M\},\ x,\ S \,\rangle \to \langle\, \Gamma,\ M,\ \#x : S \,\rangle \qquad (\textit{Lookup})$$

$$\langle\, \Gamma,\ V,\ \#x : S \,\rangle \to \langle\, \Gamma\{x = V\},\ V,\ S \,\rangle \qquad (\textit{Update})$$

$$\langle\, \Gamma\{x \stackrel{\bullet}{=} M\},\ x,\ S \,\rangle \to \langle\, \Gamma,\ M,\ S \,\rangle \quad \text{if } x \notin \mathsf{FV}(\Gamma, M, S) \qquad (\textit{Lookup-}\bullet)$$

$$\langle\, \Gamma,\ \mathsf{let}\ \Gamma'\ \mathsf{in}\ N,\ S \,\rangle \to \langle\, \Gamma\Gamma',\ N,\ S \,\rangle \qquad (\textit{Letrec})$$

$$\langle\, \Gamma,\ R[M],\ S \,\rangle \to \langle\, \Gamma,\ M,\ R : S \,\rangle \qquad (\textit{Push})$$

$$\langle\, \Gamma,\ V,\ R : S \,\rangle \to \langle\, \Gamma,\ M,\ S \,\rangle \quad \text{if } R[V] \rightsquigarrow M \qquad (\textit{Reduce})$$

$$(\lambda x.M)\, y \rightsquigarrow M\,[y/x]$$

$$\mathsf{case}\ c_j\, \vec{y}\ \mathsf{of}\ \{c_i\, \vec{x}_i \to M_i\} \rightsquigarrow M_j[\vec{y}/\vec{x}_j]$$

$$m + N \rightsquigarrow \mathsf{add}_m N$$

$$\mathsf{add}_m n \rightsquigarrow \ulcorner m + n \urcorner$$

$$\mathsf{iszero}\, m \rightsquigarrow \begin{cases} \mathsf{true} & \text{if } m = 0 \\ \mathsf{false} & \text{otherwise} \end{cases}$$

Figure 1: Abstract machine semantics

Definition 2.1 (GC)
Garbage collection can be applied to a closed configuration Σ to obtain Σ', written $\Sigma \gg \Sigma'$ if and only if Σ' is closed, and can be obtained from Σ by removing zero or more bindings and update markers from the heap and the stack respectively.

This is an accessibility-based definition as found in e.g., the gc-reduction rule of [MH98]. The removal of update-markers from the stack is not surprising given that they are viewed as the binding occurrences of the variables in question.

Transition Rules The abstract machine semantics is presented in Figure 1; we implicitly restrict the definition to well-formed closed configurations.

The first group of rules are the standard call-by-need rules. Rules (*Lookup*) and (*Update*) concern evaluation of variables. To begin evaluation of x, we remove the binding $x = M$ from the heap and start evaluating M, with x, marked for update, pushed onto the stack. Rule (*Update*) applies when this evaluation is finished, and we may update the heap with the new binding for x.

The rule for "use-once-don't-drag" bindings looks up the binding without pushing an update marker. Thus the binding will never be updated and cannot be used again. The side condition $x \notin \mathsf{FV}(\Gamma, M, S)$ enforces that their are no "dragging" references to x. Note that if there is a reference to x in a part

of the configuration which is garbage we may still apply the rule if we first remove the garbage. If the side condition cannot be fulfilled the computation gets *stuck*. Note that the computation may get stuck due to a "use-once-don't-drag" binding even though the binding is not used more than once. For example, $\mathsf{let}\ \{x \stackrel{\bullet}{=} 1+2\}\ \mathsf{in}\ x+(\lambda y.1)\,x$ gets stuck since when x is used there is a remaining (semantically dead) occurrence of x in $(\lambda y.1)\,x$ which cannot be removed by garbage collection.

Rule (*Letrec*) adds a set of bindings to the heap. Note that it is an implicit condition to keep the configuration well formed so the domain of Γ' must be fresh, i.e, $\mathrm{dom}\,\Gamma' \cap \mathrm{dom}(\Gamma, S) = \emptyset$. This condition can always be satisfied by a local α-conversion.

The basic computation rules are captured by the (*Push*) and (*Reduce*) rules schemas. The rule (*Push*) allows us to get to the heart of the evaluation by "unwinding" a shallow reduction context. When the term to be evaluated is a value and there is a reduction context on the stack, the (*Reduce*) rule is applied.

2.2 Measuring time and space

We measure time consumption of a computation simply by counting the number of transitions. In practice different transitions takes different amount of time. For example, the allocation of a closure in the heap typically takes more time the more free variables of the closure. But our measure is within a constant factor (depending only on the program size) of the actual time.

Measuring space is more subtle. A desired property of a model of space-use is that it is true to actual implementations. Unfortunately, different abstract machines and garbage collection strategies differ in their asymptotic space behaviour. Although we will choose a particular model of space use we believe that the results in this paper can be adapted to any reasonable model of space use. We will use the space model from [GS99]. In [GS01b] we discuss the subtle ways in which different abstract machines and implementations described in the literature differ from this model and each other. Bakewell and Runciman [BR00a] focus on techniques for comparing different evaluators.

We measure the heap space occupied by a configuration by counting the number of bindings in the heap and the number of update markers on the stack. We count update markers on the stack as also occupying heap space, since in a typical implementation an update marker refers to a so-called "blackhole closure" in the heap – a placeholder where the update eventually will take place. We will count every binding as occupying one unit of space.

In practice the size of a binding varies since a binding is typically represented by a tag or a code pointer plus an environment with one entry for every free variable. However, the right hand side of every binding is a (possibly renamed) subexpression of the original program, (a property of the semantics sometimes called *semi-compositionality*) so counting it as occupying one unit of space gives a measure which is within a constant factor (depending only on the program

size) of the actual space used. Integers are an exception to this claim, but recall that our integers are bounded so they can also be represented in a constant amount of space.

We measure stack space by simply counting the number of elements on the stack, so an update marker will be viewed as occupying both heap and stack space. In practice every element on the stack does not occupy the same amount of space, but again, semi-compositionality of the abstract machine assures that our measure is within a program-size-dependent constant factor. The size of a configuration, written $|\Sigma|$ is a pair (h, s) where h and s is the amount of heap and stack respectively occupied by the the configuration.

We are now ready to define what it means for a computation to complete in n steps within a certain fixed amount of space.

Definition 2.2 (Convergence in fixed space)

1. $\Sigma \rightarrow_{(h,s)} \Sigma' \stackrel{\text{def}}{=} \Sigma \rightarrow \Sigma'$ and $|\Sigma| \leq (h, s)$.

2. $\twoheadrightarrow_{(h,s)} \stackrel{\text{def}}{=} \rightarrow_{(h,s)}^{*}$

3. $\Sigma \Downarrow^{n}_{(h,s)} \stackrel{\text{def}}{=} \exists \Delta, V. \Sigma \twoheadrightarrow^{n}_{(h,s)} \langle\, \Delta,\ V,\ \epsilon \,\rangle$ and $|\langle\, \Delta,\ V,\ \epsilon \,\rangle| \leq (h, s)$.

4. $M \Downarrow^{n}_{(h,s)} \stackrel{\text{def}}{=} \langle\, \emptyset,\ M,\ \epsilon \,\rangle \Downarrow^{n}_{(h,s)}$.

We read $M \Downarrow^{n}_{(h,s)}$ as M can converge in n steps within (h, s) space, i.e., the maximum heap, and stack is less than or equal to h and s respectively. Note that, with this definition, if a binding is garbage collected immediately after it has been allocated it does not account for any space. In real implementations the binding would of course momentarily take up one unit of space even if it is garbage collected immediately. However, our model is correct within a constant factor.

We will finish this section by stating a relationship between terms with "use-once-don't-drag" bindings and their unannotated counterpart (which we will denote by $\hat{M}$. If a term does not get stuck due to a "use-once-don't-drag" binding then its time and space behaviour is closely coupled to the term obtained by removing the "use-once-don't-drag" annotations.

Proposition 2.3

- $M \Downarrow^{n}_{(h,s)} \Longrightarrow \hat{M} \Downarrow^{n}_{(h,s)}$ and

- if $M \Downarrow$ then, $\hat{M} \Downarrow^{n}_{(h,s)} \Longrightarrow M \Downarrow^{\leq n}_{(h,s)}$.

The proof is straightforward since the computations are lockstep, and whenever the computation of M applies a "use-once-don't-drag" lookup step, the fact that M does not become stuck implies that in the corresponding lookup step in $\hat{M}$, the update marker can be immediately garbage collected.

3 Notions of Work and Space Safety

In this section we define two different notions of work and space safe transformations – a weak asymptotic notion and a strong absolute one.

The starting point for an operational theory is usually an approximation and an equivalence defined in terms of *program contexts*. Program contexts are usually introduced as "programs with holes", the intention being that an expression is to be "plugged into" all of the holes in the context. The central idea is that to compare the behaviour of two terms one should compare their behaviour in all program contexts.

We will use contexts such that holes may not occur in argument positions of an application or a constructor, for if this were the case, then filling a hole (with a non variable) would violate the syntax. Contexts may contain zero or more occurrences of the hole, and as usual the operation of filling a hole with a term can cause variables in the term to become captured. We will use $\mathbb{C}$ and $\mathbb{D}$ to range over contexts. The grammar of contexts is as follows.

$$\begin{array}{rcl} \mathbb{C},\mathbb{D} & ::= & [\cdot] \mid x \mid \lambda x.\mathbb{C} \mid \mathbb{C}\, x \mid c\,\vec{x} \\ & \mid & n \mid \mathbb{C} + \mathbb{D} \mid \mathsf{add}_n\mathbb{C} \mid \mathsf{iszero}\,\mathbb{C} \\ & \mid & \mathsf{let}\ \{\vec{\mathbb{B}}\}\ \mathsf{in}\ \mathbb{D} \mid \mathsf{case}\ \mathbb{C}\ \mathsf{of}\ \{c_i\,\vec{x}_i \to \mathbb{D}_i\} \\ \mathbb{B} & ::= & x = \mathbb{C} \mid x \stackrel{\bullet}{=} \mathbb{C} \end{array}$$

We will write $\mathsf{CV}(\mathbb{C})$ for the variables that may be captured when filling the holes in $\mathbb{C}$.

We are now ready to define our two different notions of work and space safety based on *improvement theory* which was first developed in the call-by-name setting [San95, San91, San96] for the purpose of reasoning about running-times of programs. The two notions are the conjunctions of the two corresponding notions of work-safety of call-by-need in [MS99a] and space-safety of call-by-need in [GS99].

3.1 Weak Improvement

The first definition is a weak asymptotic notion of work and space safety which we will refer to as weak improvement.

Definition 3.1 (Weak Improvement)
We say that M *is weakly improved by* N, written $M \underset{\approx}{\rhd} N$, if there exists a linear function $f \in \mathbb{N} \to \mathbb{N}$ such that for all $\mathbb{C}$, σ such that $\mathbb{C}[M\sigma]$ and $\mathbb{C}[N\sigma]$ are closed,

$$\mathbb{C}[M\sigma]{\Downarrow}^{n}_{(h,s)} \implies \mathbb{C}[N\sigma]{\Downarrow}^{\leq f(n)}_{(f(h),f(s))}.$$

So $M \underset{\approx}{\rhd} N$ means that N never takes up more than a constant factor more time or space than M (but it might still use non-constant factor less time and space).

This notion of work and space safety has been criticised by Minamide [Min00] because if we repeatedly apply a transformation step which is a weak improvement the constant factor may multiply up: suppose that we are transforming a program fragment of size n and suppose we perform n transformation steps. If each step may double the time and space required then the transformation sequence may increase the the time and space required by 2^n. Another problem with weak improvement is that it is semantically badly behaved – in [GS01b] it is shown that weak space improvement is discontinuous with respect to unfolding of recursive definitions and that the natural context lemma fails.

3.2 Strong Improvement

An alternative notion of work and space safety is a strong absolute notion of space improvement.

Definition 3.2 (Strong improvement)
M is *strongly improved by* N, written $M \gtrsim N$, if for all $\mathbb{C}$, σ such that $\mathbb{C}[M\sigma]$ and $\mathbb{C}[N\sigma]$ are closed,

$$\mathbb{C}[M\sigma]{\Downarrow}^{n}_{(h,s)} \implies \mathbb{C}[N\sigma]{\Downarrow}^{\leq n}_{(h,s)}.$$

We write $M \lesssim\!\gtrsim N$ to mean that $M \gtrsim N$ and $N \gtrsim M$. Although the definition of strong improvement is somewhat arbitrary – since it deals with constant factors for a high-level abstract machine it has the property that an arbitrary number of transformation steps can be performed without any constant factor multiplying up. Significant for this paper is also that we can show that inlining of "use-once-don't-drag" bindings is a strong improvement which implies that it is also a weak improvement.

Our main technical vehicle for showing strong improvements is a *context lemma* [Mil77]: to prove that M is strongly improved by N, one only needs to compare their behaviour with respect to a much smaller set of contexts, namely the context which immediately need to evaluate their holes.

Lemma 3.3 (Context Lemma)
For all M and N such that $\mathsf{FV}(M) \supseteq \mathsf{FV}(N)$, if for all Γ, S and σ,

$$\langle \Gamma,\ M\sigma,\ S \rangle{\Downarrow}^{n}_{h,s} \implies \langle \Gamma,\ N\sigma,\ S \rangle{\Downarrow}^{\leq n}_{h,s}$$

then $M \gtrsim N$.

The lemma is essentially the conjunction of the context lemma for strong time improvement in [MS99a] and the context lemma for strong space improvement in [GS99]. However there is one subtlety. Because of the "use-once-don't-drag" bindings, successful termination can depend on free variables so the context lemma as stated in [MS99a] is not valid for our language. The key extra premise

is $\mathsf{FV}(M) \supseteq \mathsf{FV}(N)$ which guarantees that N cannot make the computation get stuck unless M can already. In the context lemma for space improvement [GS99] the extra premise is there already because extra references can hold on to extra space. This is reflected in the proof of the context lemma as stated here – it is a slight adaption of the proof for the context lemma in [GS99]. The proof is rather involved and we will not reproduce it here but refer the reader to [MS99b] and [GS01b].

4 Work and Space Safe Inlining

In this section we show the work and space safety of β-reduction for a "use-once-don't-drag" application, $(\lambda x.M) \bullet N$. Recall that we consider $(\lambda x.M) \bullet N$ as syntactic sugar for $\mathsf{let}\ \{y \stackrel{\bullet}{=} N\}\ \mathsf{in}\ (\lambda x.M)\, y$ where $y \stackrel{\bullet}{=} N$ is a "use-once-don't-drag" binding. This makes it possible to divide the problem into two steps. The first step is a restricted β-reduction transformation

$$(\lambda x.M)\, y \Rightarrow M[{}^{y}/_{x}].$$

In [MS99a] this transformation was shown to be a strong time improvement and in [GS99] to be a strong space improvement. The second step is validated by the following theorem.

Theorem 4.1
If $\mathsf{CV}(\mathbb{C}) \cap (\mathsf{FV}(M) \cup \{y\}) = \emptyset$ then

$$\mathsf{let}\ \{y \stackrel{\bullet}{=} M\}\ \mathsf{in}\ \mathbb{C}[y] \gtrsim \mathsf{let}\ \{y \stackrel{\bullet}{=} M\}\ \mathsf{in}\ \mathbb{C}[M]$$

Proof. The context lemma makes the proof straightforward because it is enough to show that for all Γ_0, S_0 and σ,

$$\begin{array}{l} \langle\, \Gamma_0,\ (\mathsf{let}\ \{y \stackrel{\bullet}{=} M\}\ \mathsf{in}\ \mathbb{C}[y])\sigma,\ S_0\,\rangle \Downarrow^{n}_{(h,s)} \\ \Longrightarrow \\ \langle\, \Gamma_0,\ (\mathsf{let}\ \{y \stackrel{\bullet}{=} M\}\ \mathsf{in}\ \mathbb{C}[M])\sigma,\ S_0\,\rangle \Downarrow^{\leq n}_{(h,s)} \end{array}$$

Thus assume

$$\langle\, \Gamma_0,\ (\mathsf{let}\ \{y \stackrel{\bullet}{=} M\}\ \mathsf{in}\ \mathbb{C}[y])\sigma,\ S_0\,\rangle \Downarrow^{n}_{(h,s)}$$

i.e.,

$$\begin{array}{ll} & \langle\, \Gamma_0,\ (\mathsf{let}\ \{y \stackrel{\bullet}{=} M\}\ \mathsf{in}\ \mathbb{C}[y])\sigma,\ S_0\,\rangle \\ \rightarrow_{(h,s)} & \langle\, \Gamma_0\{y \stackrel{\bullet}{=} M\sigma\},\ \mathbb{C}[y]\sigma,\ S_0\,\rangle \\ \gtrdot & \langle\, \Gamma_1\{y \stackrel{\bullet}{=} M\sigma\},\ \mathbb{C}[y]\sigma,\ S_1\,\rangle \Downarrow^{n-1}_{(h,s)} \end{array}$$

We have assumed, without loss of generality, that σ acts as the identity on the bound variables in the configuration. We have also assumed that there is at

least one hole in $\mathbb{C}$ which guarantees that the binding for y cannot be garbage collected. If there is no hole in $\mathbb{C}$ the left and right hand side of the improvement coincide so the statement is then trivially true. It suffices to show that

$$\begin{array}{l}\langle\, \Gamma_1\{y \stackrel{\bullet}{=} M\sigma\},\ \mathbb{C}[y]\sigma,\ S_1\,\rangle\Downarrow^{n-1}_{(h,s)}\\ \Longrightarrow\\ \langle\, \Gamma_1\{y \stackrel{\bullet}{=} M\sigma\},\ \mathbb{C}[M]\sigma,\ S_1\,\rangle\Downarrow^{\leq n-1}_{(h,s)}\end{array}$$

because then we have

$$\begin{array}{ll} & \langle\, \Gamma,\ (\mathsf{let}\ \{y \stackrel{\bullet}{=} M\}\ \mathsf{in}\ \mathbb{C}[M])\sigma,\ S\,\rangle\\ \rightarrow_{(h,s)} & \langle\, \Gamma_0\{y \stackrel{\bullet}{=} M\sigma\},\ \mathbb{C}[M]\sigma,\ S_0\,\rangle\\ \gtrdot & \langle\, \Gamma_1\{y \stackrel{\bullet}{=} M\sigma\},\ \mathbb{C}[M]\sigma,\ S_1\,\rangle\Downarrow^{\leq n-1}_{(h,s)}\end{array}$$

as required. To show the desired implication we will show a generalised statement. In this statement we will use $\mathbb{\Gamma}$ to range over heaps with holes analogously to term contexts. Similarly $\mathbb{S}$ ranges over stacks with holes. The generalised statement is: for every m, $\mathbb{\Gamma}$, $\mathbb{C}$, $\mathbb{S}$ such that $\mathsf{CV}(\mathbb{\Gamma},\mathbb{C},\mathbb{S}) \cap (\mathsf{FV}(M) \cup \{y\}) = \emptyset$,

$$\begin{array}{l}\langle\, \mathbb{\Gamma}[y]\{y \stackrel{\bullet}{=} M\sigma\},\ \mathbb{C}[y],\ \mathbb{S}[y]\,\rangle\Downarrow^{m}_{(h,s)}\\ \Longrightarrow\\ \langle\, \mathbb{\Gamma}[M]\{y \stackrel{\bullet}{=} M\sigma\},\ \mathbb{C}[M],\ \mathbb{S}[M]\,\rangle\Downarrow^{\leq m}_{(h,s)}.\end{array}$$

The proof is by induction over m and is easy since the two computation are lockstep apart from the step (if it takes place) in the first computation that looks up y. □

4.1 Work and Space-Safety for Usage Analysis

It is perhaps not immediately apparent how the notion of "use-once-don't-drag" binding and Theorem 4.1 can be used to argue the work and space safety of usage analysis so we will spell it out in this section.

Usage analysis are global program analyses; they can take the context in which a term occurs into account. Not surprisingly the results that have been established for usage analyses involve the whole program.

Proposition 4.2
If P is a program (a closed term with no "use-once-don't-drag" bindings) and P' is obtained from P by replacing bindings with "use-once-don't-drag" bindings whenever one of the usage analyses of [TWM95, Gus98, Gus99, WPJ99, WPJ00, GS00] claims the binding is "use-once" then

$$P\Downarrow \implies P'\Downarrow.$$

The proofs of this claim vary in nature for the different analyses but it essentially amounts to the subject reduction property of the respective type systems, which implies that well-typed programs cannot become stuck due to the configuration becoming open. [The latter point is only proved explicitly in [Gus98, Gus99, GS00]. In [TWM95] and [WPJ99] the result is established for the weaker notion of "used at most once" but we believe it is straightforward to strengthen their results.]

The property that we wish to prove is a similarly global property, rather than a context-insensitive improvement relation:

Theorem 4.3
If P is a program (a closed term with no "use-once-don't-drag" bindings), such that

$$P \Downarrow^{n}_{(h,s)},$$

and Q is obtained from P by inlining some of the bindings which are "use-once" according to one of the analyses mentioned in Proposition 4.2, then

$$Q \Downarrow^{\leq n}_{(h,s)}.$$

PROOF. Suppose that $P \Downarrow^{n}_{(h,s)}$, and that P' is the result of replacing all bindings which are "use-once" according to one of the type systems with actual "use-once-don't-drag" bindings. Suppose further that Q' is the result of inlining some of these bindings in P', and that Q is the result of removing all "use-once-don't-drag" annotations from Q'. From the soundness of the respective analysis (Proposition 4.2) we know that $P' \Downarrow$ so by proposition 2.3 $P' \Downarrow^{\leq n}_{(h,s)}$. Now since Q' is obtained from P' by inlining "use-once-don't-drag" bindings, from Theorem 4.1 and the definitions of improvement, it follows that $Q' \Downarrow^{\leq n}_{(h,s)}$. Finally, since $\hat{Q'} = Q$, by proposition 2.3 we have $Q \Downarrow^{\leq n}_{(h,s)}$ as required. □

5 Related Work

This paper relies heavily on the work by Moran and Sands on time improvement for call-by-need [MS99a] and the work by Gustavsson and Sands on space improvement for call-by-need [GS99, GS01a, GS01b]. Improvement theory was first developed in the call-by-name setting [San95, San91, San96] for the purpose of reasoning about running-times of programs. Minamide [Min00] suggests an alternative to our definition of improvement based on additive constant factors but its properties are not studied for any particular language.

Other related work includes the development of "space-aware" operational models for call-by-need languages [Ses97, Ros96, BLR96, BR00b], studies of space-safety properties of global transformations [Min99, Min00] and of the relative efficiency of different abstract machines [BG96, Cli98, BR00a, Min00].

Morrisett and Harper [MH98] use a similar style of abstract machine description to that used here in order to investigate the semantics of memory management in an ML-like language (see also [MFH95]). They give abstract specifications of garbage collection, and prove the correctness of a particular type-based collection scheme.

A number of insights into space problems of lazy evaluation – which we have found useful – can be found in a range of sources, e.g., [Jon92, Wad87, Spa93, PJ87, Hug83, RW93, Röj95]

The idea of usage analysis is old and goes back at least to Fairbairn [Fai85] but he gives no analysis. As far as we know Fairbairn and Wray were the first to report on a simple local usage analysis which was used to avoid pushing update markers in the Three Instruction Machine [FW87]. The first non local analyses, that we are aware of, that can provide usage information are a backwards abstract interpretation by Hughes and Wray [Hug88] and a path (evaluation order) analysis by Bloss, Hudak and Young [BHY88]. The refined path analysis by Gomard and Sestoft [GS91] can also provide usage information. In his PhD thesis Sestoft presents a so called usage interval analysis [Ses91] which can give a lower and an upper bound on the number of times an expression is used under call-by-name. The first type based usage analysis is due to Launchbury, Gill, Hughes, Marlow, Peyton Jones and Wadler [LGH+92] and it incorporates ideas from linear logic (as proposed by Abramsky [Abr90, Abr93]). Marlow presents an analysis based on abstract interpretation [Mar93] and Faxén [Fax95] and Boquist and Johnsson [BJ96] formulates usage analyses based on flow analysis. The first usage analysis to be argued correct is the type based analysis by Turner, Wadler and Mossin [TWM95] which was proved correct with respect to a notion of "use-once" bindings. However, we belive that it is easy to strengthen their result to "use-once-don't-drag" bindings. Mogensen takes the type system by Turner et al as his starting point and adopts it by a notion of zero usage [Mog97]. Thereby it provides more accurate information about "use-once" bindings but the analysis is not sound with respect to "use-once-don't-drag". The analysis by Turner et al has been extended with usage subtyping by Gustavsson [Gus98, Gus99] and Wansbrough and Peyton Jones [WPJ99] and with different degrees of usage polymorphism by Wansbrough and Peyton Jones [WPJ00] and Gustavsson and Svenningsson [GS00]. The analyses in [Gus98, Gus99, GS00] have been proved to satisfy the "use-once-don't-drag" criteria and we believe the analyses in [WPJ99, WPJ00] do so as well.

6 Conclusions and Future Work

A number of researchers, e.g., Turner, Wadler and Mossin [TWM95], have sought to find criteria for when β-reduction in a call-by-need language is work-safe, based on notions of "used at most once". Despite the fact that Turner et al discuss inlining of "use-once" bindings in some detail, as far as we are aware, no

usage analysis have previously been proved to guarantee work-safety. Another question (one which to our knowledge has not even been posed) is whether the "used at most once" criteria might also guarantee space safety. In this paper we have shown that the "used at most once" criteria is not enough to guarantee space-safety. We therefore strengthened the use-once criteria and showed that the stronger criteria of "use-once-don't-drag" is enough to guarantee both work and space safety. Some of the published usage analyses, including the analysis by Turner et al, satisfy the stronger criteria so work and space safety follows from our result. Some other analyses, e.g. [Mog97], do not satisfy the additional criteria, and we believe that as a result those analyses do not provide conditions for space-safe inlining.

An issue for future work is whether the intuitive "used-at-most" criteria can guarantee work-safety of β-reduction even though it doesn't guarantee space safety. We believe that it could be shown quite direct along the lines of the proof in this paper. Another issue is the let-floating transformation

$$\mathsf{let}\ \{x = M\}\ \mathsf{in}\ \lambda y.N \Rightarrow \lambda y.\mathsf{let}\ \{x = M\}\ \mathsf{in}\ N.$$

This transformation step is not space nor work safe in general because it may duplicate the computation of M. But what if the abstraction is used at most once? The analysis by Turner et al and its followups can provide such information but as far as we know it remains an open problem to show that the transformation then would be work and space safe.

References

[Abr90] Samson Abramsky. Computational interpretations of linear logic. Technical Report DOC 90/20, Imperial College, Department of Computing, 1990.

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[BG96] Guy E. Blelloch and John Greiner. A provably time and space efficient implementation of nesl. In *Proc. ICFP'96*, pages 213–225, 1996.

[BHY88] A. Bloss, P. Hudak, and J. Young. Code optimisations for lazy evaluation. *Lisp and Symbolic Computation*, 1:167–164, September 1988.

[BJ96] U. Boquist and T. Johnsson. The grin project: A highly optimising back end for lazy functional languages. In *Proc. of IFL'96, Bad Godesberg, Germany*. Springer Verlag LNCS 1268, 1996.

[BLR96] Z.-E.-A. Benaissa, P. Lescanne, and K. H. Rose. Modeling sharing and recursion for weak reduction strategies using explicit substitution. In *Proc. PLILP'96*, volume 1140 of *LNCS*, pages 393–407. Springer-Verlag, 1996.

[BR00a] Adam Bakewell and Colin Runciman. A model for comparing the space usage of lazy evaluators. In *Proceedings of Principles and Practice of Declarative Programming*, September 2000.

[BR00b] Adam Bakewell and Colin Runciman. A space semantics for core haskell. In *Proceedings of the Haskell Workshop*, September 2000.

[Cli98] William D. Clinger. Proper tail recursion and space efficiency. In *Proc. PLDI'98*, 1998.

[Fai85] Jon Fairbairn. Removing Redundant Laziness from Supercombinators. In *Workshop on Implementation of Functional Languages*, pages 181–189. Programming Methodology Group Chalmers University of Technology. PMG Report 17, 1985.

[Fax95] Karl-Filip Faxén. Optimizing lazy functional programs using flow inference. In *Second International Symposium on Static Analysis*, pages 136–153. Springer-Verlag, LNCS 983, 1995.

[FW87] Jon Fairbairn and Stuart Wray. TIM: A Simple, Lazy Abstract Machine to Execute Supercombinators. In *IFIP conference on Functional Programming Languages and Computer Architecture*, pages 34–45. Springer Verlag LNCS 274, 1987.

[GS91] Carsten K. Gomard and Peter Sestoft. Evaluation Order Analysis for Lazy Data Structures. In *Proc. 1991 Glasgow Workshop on Functional Programming*, Workshops in Computing. Springer–Verlag, 1991.

[GS99] J. Gustavsson and D. Sands. A foundation for space-safe transformations of call-by-need programs. In A. D. Gordon and A. M.Pitts, editors, *The Third International Workshop on Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999.

[GS00] J. Gustavsson and J. Svenningsson. A usage analysis with bounded usage polymorphism and subtyping. In *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, pages 140–157. Springer-Verlag, LNCS 2011, September 2000.

[GS01a] J. Gustavsson and D. Sands. Possibilities and limitations of call-by-need space improvement. In *Proceedings of the International Conference on Functional Programming*, September 2001. To Appear.

[GS01b] J. Gustavsson and D. Sands. Space-safe transformations of call-by-need programs. Paper I in this thesis, May 2001.

[Gus98] J. Gustavsson. A type based sharing analysis for update avoidance and optimisation. In *Proc. ICFP'98*, pages 39–50, September 1998.

[Gus99] Jörgen Gustavsson. A Type Based Sharing Analysis for Update Avoidance and Optimisation. Licentiate thesis, May 1999.

[Hug83] R. J. M. Hughes. *The Design and Implementation of Programming Languages*. PhD thesis, Programming Research Group, Oxford University, July 1983.

[Hug88] J. Hughes. Backwards Analysis of Functional Programs. In Bjørner and Ershov, editors, *IFIP Workshop on Partial Evaluation and Mixed Computation*, pages 187–208, 1988.

[Jon92] Richard Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. POPL'93*, pages 144–154. ACM Press, January 1993.

[LGH+92] J. Launchbury, A. Gill, J. Hughes, S. Marlow, S. L. Peyton Jones, and P. Wadler. Avoiding Unnecessary Updates. In J. Launchbury and P. M. Sansom, editors, *Functional Programming*, Workshops in Computing, Glasgow, 1992.

[Mar93] S. Marlow. Update avoidance analysis by abstract interpretation. In *Proc. 1993 Glasgow Functional Programming Workshop*, Workshops in Computing. Springer-Verlag, August 1993.

[MFH95] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract Models of Memory Management. In *Proc. FPCA'95*, pages 66–77. ACM Press, June 1995.

[MH98] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 175–226. Cambridge University Press, 1998.

[Mil77] R. Milner. Fully abstract models of the typed λ-calculus. *Theoretical Computer Science*, 4:1–22, 1977.

[Min99] Yasuhiko Minamide. Space-profiling semantics of the call-by-value lambda calculus and the cps transformation. In *Proc. HOOTS III*, 1999. To appear as ENTCS.

[Min00] Yasuhiko Minamide. A new criterion for safe program transformations. 2000. To appear as ENTCS.

[Mog97] T. Mogensen. Types for 0, 1 or many uses. In *Proceedings of IFL '97: 9th International Workshop on Implementation of Functional Languages*, pages 112–122, St. Andrews, Scotland, September 1997. Springer-Verlag, LNCS 1467.

[MS99a] Andrew Moran and David Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Proc. POPL'99*, pages 43–56. ACM Press, January 1999.

[MS99b] Andrew Moran and David Sands. Improvement in a lazy context: An operational theory for call-by-need (extended version). Extended version of [MS99a], 1999.

[PJ87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[PJPS96] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proc. ICFP'96*, pages 1–12. ACM Press, May 1996.

[PJS98] S. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998.

[Röj95] Niklas Röjemo. *Garbage collection, and memory efficiency, in lazy functional languages*. PhD thesis, Chalmers Tekniska Högskola, 1995.

[Ros96] K. H. Rose. *Operational Reduction Models for Functional Programming Languages*. PhD thesis, DIKU, University of Copenhagen, Denmark, February 1996. available as DIKU report 96/1.

[RW93] Colin Runciman and David Wakeling. Heap Profiling of Lazy Functional Programs. *Journal of Functional Programming*, 3(2):217–245, April 1993.

[San91] D. Sands. Operational theories of improvement in functional languages (extended abstract). In *Proc. 1991 Glasgow Functional Programming Workshop*, Workshops in Computing Series, pages 298–311. Springer-Verlag, August 1991.

[San95] D. Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995.

[San96] D. Sands. Total correctness by local improvement in the transformation of functional program. *ACM TOPLAS*, 18(2):175–234, March 1996.

[Ses91] P. Sestoft. *Analysis and Efficient Implementation of Functional Programs*. PhD thesis, DIKU, University of Copenhagen, Denmark, October 1991.

[Ses97] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[Spa93] Jan Sparud. Fixing Some Space Leaks without a Garbage Collector. In *Proc. 6th Int'l Conf. on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 117–122. ACM Press, June 1993.

[TWM95] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proc. FPCA'95*, pages 1–11. ACM Press, June 1995.

[Wad87] P. Wadler. Fixing Some Space Leaks with a Garbage Collector. *Software Practice and Experience*, September 1987.

[WPJ99] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *Proc. POPL'99*. ACM Press, January 1999.

[WPJ00] Keith Wansbrough and Simon Peyton Jones. Simple Usage Polymorphism. In *ACM SIGPLAN Workshop on Types in Compilation*, September 2000.

Paper III

A Type Based Sharing Analysis for Update Avoidance and Optimisation

Abstract

Sharing of evaluation is crucial for the efficiency of lazy functional languages, but unfortunately the machinery to implement it carries an inherent overhead. In abstract machines this overhead shows up as the cost of performing updates, many of them actually unnecessary, and also in the cost of the associated bookkeeping, that is keeping track of when and where to update. In spineless abstract machines, such as the STG-machine and the TIM, this bookkeeping consists of pushing, checking for and popping update markers. Checking for update markers is a very frequent operation and indeed the implementation of the STG-machine has been optimised for fast update marker checks at the expense of making the pushing and popping of update markers more costly.

In this thesis we present a type based sharing analysis that can determine when updates can be safely omitted and marker checks bypassed. The type system is proved sound with respect to the lazy Krivine machine and enjoys a principal typing property. We have implemented the analysis and the preliminary benchmarks seem very promising. Most notably, virtually all update marker checks can be avoided. This may make the tradeoffs of current implementations obsolete and calls for new abstract machine designs.

Contents

Chapter 1

Introduction

1.1 Sharing of evaluation

In a call-by-name functional language arguments to functions are passed unevaluated. For example, $(\lambda x.x + x)\ (1 + 2)$ evaluates as follows.

$$(\lambda x.x + x)\ (1 + 2) \mapsto (1 + 2) + (1 + 2) \mapsto 3 + (1 + 2) \mapsto 3 + 3 \mapsto 6$$

Note that in the first step the argument $1 + 2$ is duplicated with the effect that $1 + 2$ is computed twice. This duplication of computation is devastating for the efficiency of call-by-name languages. Therefore non-strict functional languages usually rely on a call-by-need semantics where the evaluation of arguments is shared between different uses so that an argument is computed at most once.

Although this sharing of evaluation is crucial for the efficiency of lazy languages, it also carries a substantial run time overhead. This overhead and how to reduce it is the subject of this thesis. This work has been previously reported on in [Gus98].

1.2 The overhead of sharing

In the implementation of a lazy functional language sharing of evaluation is performed by updating. For example, the evaluation of $(\lambda x.x + x)\ (1 + 2)$ proceeds as follows. First, a closure for $1 + 2$ is built in the heap and a reference to the closure is passed to the abstraction. Second, to evaluate $x + x$ the value of x is required. Thus the closure is fetched from the heap and evaluated. Third, the closure is updated (ie overwritten) with the result, so that when the value of x is required again the expression needs not be recomputed. However, if the value of x had not been required again this update would had been wasted. This happens, for example, in the evaluation of $(\lambda x.x + 1)\ (1 + 2)$.

Measurements suggest that typically 70% of all updates are unnecessary and that about 20% of the execution time is spent on these unnecessary updates [Mar93]. It is therefore no surprise that considerable effort has been put into static analyses that can discover unnecessary updates [Ses91, LGH+92, Mar93, TWM95, Fax95, Mog97].

Besides the cost of performing updates there is also a cost associated with the bookkeeping of updates, that is keeping track of when and where to update. In the design and implementation of abstract machines, considerable attention has been given to minimising the bookkeeping associated with shared computation. See for example [PJ92]. However, comparatively little work has been done to eliminate bookkeeping overheads by static program analysis. The only work we are aware of is an analysis by Sestoft [Ses91].

In this thesis we will present a type based sharing analysis that can determine when updates can be safely omitted and also enables us to optimise the bookkeeping of updates. We will take the type system by Turner, Wadler and Mossin [TWM95] as our starting point. Our type system has a number of properties.

- It is more precise than the analysis by Turner et al, that is it will discover more unnecessary updates.
- It provides information that enable us to optimise the bookkeeping of updates. Indeed, this is our major contribution.
- It handles all features of a realistic functional language including higher order functions, data structures and mutual recursion.
- It is proved sound with respect to the lazy Krivine machine, by showing that evaluation preserves typings.
- Preliminary benchmarks indicates that it is surprisingly effective.

1.3 Reducing the overhead of sharing

We can reduce the overhead of sharing if we can discover unnecessary updates. Consider the following program.

$$\begin{array}{l}\texttt{let}\ x = 1 + 2\ \texttt{in}\\ \texttt{let}\ y = (\lambda z.z)\ x\ \texttt{in}\\ y + y\end{array}$$

Here, the value of y is clearly needed twice. Thus the closure referred to by y needs to be updated so that $(\lambda z.z)\ x$ gets computed only once. Since $(\lambda z.z)\ x$ gets computed only once, x will be dereferenced only once and therefore it is

unnecessary to update the closure referenced by x. An analysis can provide this information by annotating the expression as follows.

$$\begin{array}{l}\texttt{let}\ x =^{\checkmark}\ 1 + 2\ \texttt{in}\\ \texttt{let}\ y =^{!}\ (\lambda z.z)\ x\ \texttt{in}\\ y + y\end{array}$$

Here, annotating the binding of x with a $\checkmark$ indicates that the corresponding closure needs not be updated and annotating the binding of y with an ! indicates that the corresponding closure needs to be updated.

We can reduce the overhead of sharing if we can reduce the bookkeeping of updates by predicting when updates take place. Potentially, an update may be needed whenever a value has been computed. However, in our example only one update needs to take place, namely when y is updated with the result of $1 + 2$. An analysis can provide this information by annotating the expression as follows.

$$\begin{array}{l}\texttt{let}\ x =^{\checkmark}\ 1^0 +^1 2^0\ \texttt{in}\\ \texttt{let}\ y =^{!}\ (\lambda^0 z.z)\ x\ \texttt{in}\\ y +^0 y\end{array}$$

Here, the annotation 1 on $1 + 2$ indicates that exactly one closure, namely y, needs to be updated with the result of the addition. Naturally, the annotation 0 indicates that no update needs to take place. This information enables us to optimise the bookkeeping of updates and we will return to this example in section 2.2 and discuss how to apply the information in the implementation of an abstract machine.

1.4 Overview of the thesis

This thesis is organised as follows. In chapter 2 we present a small functional language and its semantics in the form of an abstract machine. We also discuss, in the context of the abstract machine, how static analysis can be used to avoid unnecessary updates and to reduce the cost of the bookkeeping of updates. We then extend the language with annotations that can express these optimisations and we give a semantics to the extended language. Based on this semantics we define the notion of a well-annotated term. In chapter 3 we present a type system in the form of a type directed translation that annotates expressions and in chapter 4 we prove the type system sound. In chapter 5 we describe the implementation, that is how to compute the well-typed annotated term that yields the best optimisation. In chapter 6 we present some experimental results from a prototype implementation. In chapter 7 we describe related work and in chapter 8, we conclude and we discuss future work.

Chapter 2

Language and semantics

In this chapter we will present a small functional language and give its semantics in the form of an abstract machine. We will also discuss how static analysis can be used to avoid unnecessary updates and to reduce the cost of the bookkeeping of updates. We will then extend the language with annotations that can express these optimisations and give a semantics to the extended language.

2.1 Language

2.1.1 Syntax

The language we use is a lambda calculus extended with integers, lists and recursive `let`-expressions. Following Launchbury [Lau93], we use a restricted syntax given below.

Variables	x, y, z		
Values	v	$::=$	$\lambda x.e \mid n \mid \texttt{nil} \mid \texttt{cons}\ x\ y$
Expressions	e	$::=$	$v \mid x \mid e\ x \mid e_0 + e_1 \mid$
			$\texttt{let}\ d\ \texttt{in}\ e \mid$
			$\texttt{case}\ e\ \texttt{of}\ alts$
Declarations	d	$::=$	$\epsilon \mid d,\ b$
Bindings	b	$::=$	$x = e$
Alternatives	$alts$	$::=$	$\{\texttt{nil} \Rightarrow e_0; \texttt{cons}\ x\ y \Rightarrow e_1\}$

The distinguishing feature of the syntax is that arguments in applications and `cons` are restricted to variables. It is straightforward to translate a term in the standard syntax into the restricted form, for example $(\lambda x.x + x)\ (1 + 2)$ is translated into $\texttt{let}\ y = 1 + 2\ \texttt{in}\ (\lambda x.x + x)\ y$. Thus the creation of a closure for $1 + 2$ is made explicit via the `let`-expression. Making the creation of closures explicit greatly simplifies the abstract machine as well as the analysis presented

in this paper. Indeed, the same restriction appears in the intermediate language of the Glasgow Haskell Compiler [JHH+93].

2.1.2 Semantics

We will take the lazy Krivine machine [Ses97] as the semantic basis of our work. The choice of an abstract machine makes the update machinery explicit and enables a soundness proof of our analysis. A correspondence between the lazy Krivine machine and Launchbury's natural semantics for lazy evaluation [Lau93] has been shown in [Ses97]. The machine can also serve as a starting point from which lower level abstract machines can be derived [Ses97].

For the purpose of the abstract machine we extend the set of terms to include expressions of the form $\texttt{add}_n\ e$. We define a reduction relation $e \mapsto e'$ between terms:

$$
\begin{array}{lcll}
(\lambda x.e)\ y & \mapsto & e[x{:=}y] & \\
n + e & \mapsto & \texttt{add}_n\ e & \\
\texttt{add}_{n_0}\ n_1 & \mapsto & n_2 \qquad \text{if } n_0 + n_1 = n_2 & \\
\texttt{case nil of} & \mapsto & e_0 & \\
\quad \texttt{nil} \Rightarrow e_0 & & & \\
\quad \texttt{cons}\ x\ y \Rightarrow e_1 & & & \\
\texttt{case cons}\ x_0\ x_1\ \texttt{of} & \mapsto & e_1[y_0{:=}x_0, y_1{:=}x_1] & \\
\quad \texttt{nil} \Rightarrow e_0 & & & \\
\quad \texttt{cons}\ y_0\ y_1 \Rightarrow e_1 & & &
\end{array}
$$

Configurations in the abstract machine are triples $\langle H\ ;\ e\ ;\ S\rangle$, where H is a heap, e is the term currently being evaluated and S is the abstract machine stack:

$$
\begin{array}{lrll}
\text{Configurations} & C & ::= & \langle H\ ;\ e\ ;\ S\rangle \\
\text{Heaps} & H & ::= & \epsilon \mid H, b \\
\text{Stacks} & S & ::= & \epsilon \mid R, S \mid \#x, S \\
\text{Reduction contexts} & R & ::= & [\,]\ x \mid [\,] + e \mid \texttt{add}_n\ [\,] \mid \\
 & & & \texttt{case}\ [\,]\ \texttt{of}\ alts
\end{array}
$$

A heap consists of a sequence of bindings. The variables bound by the heap must be distinct and the order of bindings is irrelevant. Thus a heap can be considered as a partial function mapping variables to terms and we will write $\mathrm{dom}(H)$ for the set of variables bound by H. A heap can also be considered as a declaration and vice versa. We will write H_0, H_1 for the concatenation of H_0 and H_1. An abstract machine stack is a stack of shallow reduction contexts and update markers. The stack can be thought of as corresponding to the "surrounding derivation" in a natural semantics, where the rôle of an update marker $\#x$ is to keep track of a pending update of x. The update markers on the stack will be distinct, that is there will be no more than one pending update of the same variable. We will consider an update marker as a binder and we will write $\mathrm{dom}(S)$ for the variables bound by the update markers in S. Consequently, we will require the variables bound by the stack to be distinct from the variables

$$
\begin{array}{lcll}
\langle H\ ;\ \texttt{let}\ d\ \texttt{in}\ e\ ;\ S\rangle & \overset{\text{Let}}{\longmapsto} & \langle H, d\ ;\ e\ ;\ S\rangle & \\
\langle H, x = e\ ;\ x\ ;\ S\rangle & \overset{\text{Var}}{\longmapsto} & \langle H\ ;\ e\ ;\ \#x, S\rangle & \\
\langle H\ ;\ R[e]\ ;\ S\rangle & \overset{\text{Unwind}}{\longmapsto} & \langle H\ ;\ e\ ;\ R, S\rangle & \\
\langle H\ ;\ v\ ;\ \#x, S\rangle & \overset{\text{Update}}{\longmapsto} & \langle H, x = v\ ;\ v\ ;\ S\rangle & \\
\langle H\ ;\ v\ ;\ R, S\rangle & \overset{\text{Reduce}}{\longmapsto} & \langle H\ ;\ e\ ;\ S\rangle & \text{if } R[v] \mapsto e
\end{array}
$$

Figure 2.1: Abstract machine transistion rules

bound by the heap. We will also require that configurations are closed and we will identify configurations up to α-conversion, that is renaming of the variables bound by the heap and the stack. We will also identify configurations up to garbage meaning that we may remove or add bindings to the heap as long as the configuration remains closed (that is $\langle H_0, H_1; e; S\rangle \equiv \langle H_0; e; S\rangle$ if $\langle H_0, H_1; e; S\rangle$ and $\langle H_0\ ;\ e\ ;\ S\rangle$ are closed). We will refer to this as garbage-conversion.

An initial configuration is of the form $\langle \epsilon; e; \epsilon\rangle$, where e is a closed expression. The transition rules of the abstract machine are given in figure 2.1. The rule

$$\langle H\ ;\ \texttt{let}\ d\ \texttt{in}\ e\ ;\ S\rangle \overset{\text{Let}}{\longmapsto} \langle H, d\ ;\ e\ ;\ S\rangle$$

creates new bindings in the heap. For the rule to be applied the variables bound by d must be distinct from the variables bound by H and S, that is $\mathrm{dom}(d) \cap (\mathrm{dom}(H) \cup \mathrm{dom}(S)) = \emptyset$. This condition can always be met simply by α-converting the `let`-expression. The rule

$$\langle H, x = e\ ;\ x\ ;\ S\rangle \overset{\text{Var}}{\longmapsto} \langle H\ ;\ e\ ;\ \#x, S\rangle$$

evaluates a variable x. It looks up the corresponding expression e in the heap, removes the binding, pushes an update marker for x on the stack and starts the evaluation of e. Later, if e terminates, the update marker will see to that x gets updated with the result. The removal of the binding corresponds to so called black-holing: if the evaluation of e to weak head normal form depends on x (ie x depends directly on itself) the computation will get stuck, since x is no longer bound by the heap. Note that we still consider the configuration to be closed, since x is bound by the update marker on the stack. The rule

$$\langle H\ ;\ R[e]\ ;\ S\rangle \overset{\text{Unwind}}{\longmapsto} \langle H\ ;\ e\ ;\ R, S\rangle$$

allows us to get to the heart of the evaluation by "unwinding" a shallow reduction context. When the term to be evaluated is a value the next transition depends on whether an update marker or a reduction context is on top of the

stack. To determine which rule to apply a so called update marker check is performed. If the top of the stack is an update marker the rule

$$\langle H\ ;\ v\ ;\ \#x, S\rangle \overset{\text{Update}}{\longmapsto} \langle H, x = v\ ;\ v\ ;\ S\rangle$$

applies and the heap is updated accordingly. If it is a reduction context the rule

$$\langle H\ ;\ v\ ;\ R, S\rangle \overset{\text{Reduce}}{\longmapsto} \langle H\ ;\ e\ ;\ S\rangle \quad \text{if } R[v] \mapsto e$$

applies and the value is plugged into the reduction context and a reduction can take place.

The abstract machine presented so far has a built in inefficiency shown by the following transition sequence.

$$\langle H, x = v\ ;\ x\ ;\ S\rangle \overset{\text{Var}}{\longmapsto} \langle H\ ;\ v\ ;\ \#x, S\rangle \overset{\text{Update}}{\longmapsto} \langle H, x = v\ ;\ v\ ;\ S\rangle$$

When a value is looked up with the rule Var the binding is removed from the heap and an update marker is pushed onto the stack. Then, by the rule Update, the marker is immediately popped off the stack and the binding is added to the heap again (ie an update is performed). This is indeed a common case and the abstract machine could be optimised for it by adding the synthesised rule

$$\langle H, x = v\ ;\ x\ ;\ S\rangle \overset{\text{Lookup}}{\longmapsto} \langle H, x = v\ ;\ v\ ;\ S\rangle$$

which allows a value to be looked up without pushing a marker and performing an update. Any reasonable compiler would perform this optimisation so we have included it in our implementation to make for realistic benchmarks. However, since the rule is simply synthesised from the rules Var and Update we will not consider it further.

The machine terminates when no transition rule can be applied. It may terminate for three different reasons.

- The computation terminates successfully with a value. In this case the configuration is of the form $\langle H\ ;\ v\ ;\ \epsilon\rangle$.
- A black hole is detected, that is the configuration is of the form $\langle H\,;\ x;\ S\rangle$ where $x \notin \mathrm{dom}(H)$.
- The computation goes wrong. By going wrong we mean reaching a configuration of the form $\langle H\ ;\ v\,;\ R, S\rangle$ where $R[v] \not\mapsto$. This cannot happen if we only consider well-typed terms (ie "well typed terms cannot go wrong" [Mil78]).

We define Value, Blackhole and Wrong to be the sets of terminal configurations of the different forms. We will let V, B and W range over Value, Blackhole and Wrong respectively.

2.2 The overhead - revisited

We will now discuss the overhead of sharing and how it can be reduced in the context of the abstract machine. Consider again the example from the introduction.

$$\begin{array}{l}\texttt{let}\ x = 1 + 2\ \texttt{in}\\ \texttt{let}\ y = (\lambda z.z)\ x\ \texttt{in}\\ y + y\end{array}$$

Running this program in the abstract machine yields the transition sequence given in figure 2.2. Note that we have named the configurations in the transition sequence as C_0, C_1 etc. The costs of updates shows up in a number of places.

- In the applications of the rule Var where an update marker is pushed onto the stack to record that an update shall eventually take place. This happens in the transitions $C_3 \longmapsto C_4$ and $C_6 \longmapsto C_7$.
- In the applications of the rule Update where an update marker is popped from the stack and an update takes place. This happens in the transitions $C_{11} \longmapsto C_{12}$ and $C_{12} \longmapsto C_{13}$.
- Whenever a value is in the second component of the configuration an update marker check has to be performed to decide whether an update should take place or not. This happens in the transitions $C_5 \longmapsto C_6$, $C_8 \longmapsto C_9$, $C_{10} \longmapsto C_{11}$, $C_{11} \longmapsto C_{12}$, $C_{12} \longmapsto C_{13}$, $C_{13} \longmapsto C_{14}$ and $C_{16} \longmapsto C_{17}$.

Thus, we can reduce the cost associated with the update machinery if we can avoid unnecessary updates (which also saves the cost of pushing and popping an update marker) and if we can avoid unnecessary update marker checks.

We note that update marker checks seem to be very common. Indeed, the measurements presented in chapter 6 suggest that in an unoptimised implementation update marker checks are typically about 10 times as frequent as updates. It is therefore no surprise that implementations of abstract machines tend to be optimised for fast update marker checks at the expense of a large representation of update markers, making the pushing and popping of them more expensive. For example, in the implementation of the STG-machine, update marker checks are very cheap while update markers are represented using three words where only one word (using 1 bit as a tag) would suffice [PJ92]. However, although the cost of a single update marker check is low, an analysis that can reduce the number of checks could be very worthwhile. In fact, the preliminary benchmarks of our analysis suggest that update marker checks can be avoided to such an extent that they become less frequent than updates. As a consequence the implementations of abstract machines might be the subject of review.

We can avoid an update of a closure if we can determine that there are no remaining references to the closure. In our example the update of x is

$$\langle \epsilon \,;\, \texttt{let}\, x = 1 + 2 \,\texttt{in}\, \texttt{let}\, y = (\lambda z.z)\, x \,\texttt{in}\, y + y \,;\, \epsilon \rangle \tag{C_0}$$

$$\overset{\text{Let}}{\longmapsto} \langle x = 1 + 2 \,;\, \texttt{let}\, y = (\lambda z.z)\, x \,\texttt{in}\, y + y \,;\, \epsilon \rangle \tag{C_1}$$

$$\overset{\text{Let}}{\longmapsto} \langle x = 1 + 2, y = (\lambda z.z)\, x \,;\, y + y \,;\, \epsilon \rangle \tag{C_2}$$

$$\overset{\text{Unw.}}{\longmapsto} \langle x = 1 + 2, y = (\lambda z.z)\, x \,;\, y \,;\, [\,] + y \rangle \tag{C_3}$$

$$\overset{\text{Var}}{\longmapsto} \langle x = 1 + 2 \,;\, (\lambda z.z)\, x \,;\, \#y, [\,] + y \rangle \tag{C_4}$$

$$\overset{\text{Unw.}}{\longmapsto} \langle x = 1 + 2 \,;\, \lambda z.z \,;\, [\,]\, x, \#y, [\,] + y \rangle \tag{C_5}$$

$$\overset{\text{Red.}}{\longmapsto} \langle x = 1 + 2 \,;\, x \,;\, \#y, [\,] + y \rangle \tag{C_6}$$

$$\overset{\text{Var}}{\longmapsto} \langle \epsilon \,;\, 1 + 2 \,;\, \#x, \#y, [\,] + y \rangle \tag{C_7}$$

$$\overset{\text{Unw.}}{\longmapsto} \langle \epsilon \,;\, 1 \,;\, [\,] + 2, \#x, \#y, [\,] + y \rangle \tag{C_8}$$

$$\overset{\text{Red.}}{\longmapsto} \langle \epsilon \,;\, \texttt{add}_1\ 2 \,;\, \#x, \#y, [\,] + y \rangle \tag{C_9}$$

$$\overset{\text{Unw.}}{\longmapsto} \langle \epsilon \,;\, 2 \,;\, \texttt{add}_1\ [\,], \#x, \#y, [\,] + y \rangle \tag{C_{10}}$$

$$\overset{\text{Red.}}{\longmapsto} \langle \epsilon \,;\, 3 \,;\, \#x, \#y, [\,] + y \rangle \tag{C_{11}}$$

$$\overset{\text{Upd.}}{\longmapsto} \langle x = 3 \,;\, 3 \,;\, \#y, [\,] + y \rangle \equiv \langle \epsilon \,;\, 3 \,;\, \#y, [\,] + y \rangle \tag{C_{12}}$$

$$\overset{\text{Upd.}}{\longmapsto} \langle y = 3 \,;\, 3 \,;\, [\,] + y \rangle \tag{C_{13}}$$

$$\overset{\text{Red.}}{\longmapsto} \langle y = 3 \,;\, \texttt{add}_3\ y \,;\, \epsilon \rangle \tag{C_{14}}$$

$$\overset{\text{Unw.}}{\longmapsto} \langle y = 3 \,;\, y \,;\, \texttt{add}_3\ [\,] \rangle \tag{C_{15}}$$

$$\overset{\text{Loo.}}{\longmapsto} \langle y = 3 \,;\, 3 \,;\, \texttt{add}_3\ [\,] \rangle \equiv \langle \epsilon \,;\, 3 \,;\, \texttt{add}_3\ [\,] \rangle \tag{C_{16}}$$

$$\overset{\text{Red.}}{\longmapsto} \langle \epsilon \,;\, 6 \,;\, \epsilon \rangle \tag{C_{17}}$$

Figure 2.2: A transition sequence

superfluous; no occurrence of x remains so the binding is dead. We can convey this information by annotating the expression as follows.

$$\begin{array}{l} \texttt{let}\, x =^{\checkmark} 1 + 2 \,\texttt{in} \\ \texttt{let}\, y =^{!} (\lambda z.z)\, x \,\texttt{in} \\ y + y \end{array}$$

Here, annotating the binding of x with a ✓ means that it shall not be updated and annotating the binding of y with an ! means that it shall be updated.

We can avoid update marker checks by predicting what will be on top of the stack when a value has been computed and either of the rules

$$\langle H\ ;\ v\ ;\ R, S\rangle \overset{\text{Reduce}}{\longmapsto} \langle H\ ;\ e\ ;\ S\rangle \quad \text{if } R[v] \mapsto e$$

or

$$\langle H\ ;\ v\ ;\ \#x, S\rangle \overset{\text{Update}}{\longmapsto} \langle H, x = v\ ;\ v\ ;\ S\rangle$$

may be applied. If we know that there can be no or that there must be at least one update marker on top of the stack then we can bypass the update marker check and apply the the appropriate rule directly. In our example only two updates take place, namely when x and y are updated with the result of $1 + 2$. Thus, when $1 + 2$ has been computed two update markers reside on top of the stack, one saying that x shall be updated and one saying that y shall be updated (see configuration C_{11}). However, if we bypass the update of x, as suggested above, there will only be one marker there. We can take advantage of this fact by annotating $1 + 2$ as $1 +^1 2$. The intuitive meaning of the annotation is that the compiled code for $1 +^1 2$ shall take for granted that there is exactly one update marker on the stack and pop it off the stack without performing an update marker check. In our example (the compiled code of) no other value needs to take care of any update marker and we can exploit this fact by annotating them with the annotation 0. For example $\lambda z.z$ will be annotated as $\lambda^0 z.z$, which allows us to avoid the update marker check in the transition $C_5 \longmapsto C_6$. To summarise; our example can be annotated as follows.

$$\begin{array}{l}\texttt{let}\ x =^{\checkmark}\ 1^0 +^1 2^0\ \texttt{in}\\ \texttt{let}\ y =^{!}\ (\lambda^0 z.z)\ x\ \texttt{in}\\ y +^0 y\end{array}$$

In total this saves the cost of pushing and popping an update marker, performing an update and doing seven update marker checks.

2.3 A language with annotations

In this section we will present the language of annotated expressions and its semantics.

2.3.1 Syntax

We will annotate bindings in `let`-expressions with a κ, ranging over $\{\checkmark, !\}$. Here, $\checkmark$ means the binding will not be updated and ! means that it will. In the example in the previous section we annotated values and $+$ with 0 and 1 saying that no update marker and exactly one update marker needed to be taken care of. It is however not always possible to give such a precise annotation. Thus our annotation will instead give a lower and an upper bound on the numbers of

update markers that needs to be taken care of. Thus our annotations, ranged over by ι, consists of a pair $[\eta, \xi]$. The first component η gives a lower bound on the number of markers that must reside on the stack (because the value will take them for granted) and the second component ξ gives an upper bound on the number of update markers that may reside on the stack (because the value will not look for more). Consequently, we will refer to them as the lower and the upper bound. We will let η and ξ range over $\mathbb{N} \cup \{\omega\}$. Including ω allows us to have annotations like $[0, \omega]$ meaning that an arbitrary number of update markers can be taken care of, effectively serving as an escape-hatch for the analysis. Thus, the language of the previous section is extended as follows.

Variables	x, y, z		
Values	$\tilde{v}$	$::=$	$\lambda x.\tilde{e} \mid n \mid \texttt{nil} \mid \texttt{cons}\ x\ y$
Expressions	$\tilde{e}$	$::=$	$\tilde{v}^{\iota} \mid x \mid \tilde{e}\ x \mid \tilde{e}_0 +^{\iota} \tilde{e}_1 \mid$
			$\texttt{let}\ \tilde{d}\ \texttt{in}\ \tilde{e} \mid$
			$\texttt{case}\ \tilde{e}\ \texttt{of}\ \tilde{alts}$
Declarations	$\tilde{d}$	$::=$	$\epsilon \mid \tilde{d}, \tilde{b}$
Bindings	$\tilde{b}$	$::=$	$x =^{\kappa} \tilde{e}$
Alternatives	$\tilde{alts}$	$::=$	$\{\texttt{nil} \Rightarrow \tilde{e}_0; \texttt{cons}\ x\ y \Rightarrow \tilde{e}_1\}$

We will sometimes use $\lambda^{\iota} x.e$ and $\texttt{cons}^{\iota}\ x\ y$ as syntactic sugar for $(\lambda x.e)^{\iota}$ and $(\texttt{cons}\ x\ y)^{\iota}$ respectively.

2.3.2 Semantics

The meaning of the annotations is given by modifying the abstract machine of the previous section. Again, we extend the set of (annotated) terms to include the expression $\texttt{add}^{\iota}_{n}\ \tilde{e}$. We define a reduction relation $\tilde{e} \tilde{\mapsto} \tilde{e}'$ between annotated terms:

$(\lambda^{\iota} x.\tilde{e})\ y$	$\tilde{\mapsto}$	$\tilde{e}[x{:=}y]$	
$n^{\iota'} +^{\iota} \tilde{e}$	$\tilde{\mapsto}$	$\texttt{add}^{\iota}_{n}\ \tilde{e}$	
$\texttt{add}^{\iota}_{n_0}\ n_1^{\iota'}$	$\tilde{\mapsto}$	n_2^{ι}	if $n_0 + n_1 = n_2$
$\texttt{case}\ \texttt{nil}^{\iota}\ \texttt{of}$	$\tilde{\mapsto}$	$\tilde{e}_0$	
$\quad \texttt{nil} \Rightarrow \tilde{e}_0$			
$\quad \texttt{cons}\ x\ y \Rightarrow \tilde{e}_1$			
$\texttt{case}\ \texttt{cons}^{\iota}\ x_0\ x_1\ \texttt{of}$	$\tilde{\mapsto}$	$\tilde{e}_1[y_0{:=}x_0, y_1{:=}x_1]$	
$\quad \texttt{nil} \Rightarrow \tilde{e}_0$			
$\quad \texttt{cons}\ y_0\ y_1 \Rightarrow \tilde{e}_1$			

$$
\begin{array}{lcll}
\langle \tilde{H} \,;\, \texttt{let}\, \tilde{d}\, \texttt{in}\, \tilde{e} \,;\, \tilde{S} \rangle & \overset{\text{Let}}{\tilde{\longmapsto}} & \langle \tilde{H}, \tilde{d} \,;\, \tilde{e} \,;\, \tilde{S} \rangle & \\
\langle \tilde{H}, x =^{!} \tilde{e} \,;\, x \,;\, \tilde{S} \rangle & \overset{\text{Var-!}}{\tilde{\longmapsto}} & \langle \tilde{H} \,;\, \tilde{e} \,;\, \#x, \tilde{S} \rangle & \\
\langle \tilde{H}, x =^{\checkmark} \tilde{e} \,;\, x \,;\, \tilde{S} \rangle & \overset{\text{Var-}\checkmark}{\tilde{\longmapsto}} & \langle \tilde{H} \,;\, \tilde{e} \,;\, \tilde{S} \rangle & \\
\langle \tilde{H} \,;\, \tilde{R}[\tilde{e}] \,;\, \tilde{S} \rangle & \overset{\text{Unwind}}{\tilde{\longmapsto}} & \langle \tilde{H} \,;\, \tilde{e} \,;\, \tilde{R}, \tilde{S} \rangle & \\
\langle \tilde{H} \,;\, \tilde{v}^{[\eta,\xi]} \,;\, \#x, \tilde{S} \rangle & \overset{\text{Update}}{\tilde{\longmapsto}} & \langle \tilde{H}, x =^{!} \tilde{v}^{[\eta,\xi]} \,;\, \tilde{v}^{[\eta-1,\xi-1]} \,;\, \tilde{S} \rangle & \text{if } \xi \geq 1 \\
\langle \tilde{H} \,;\, \tilde{v}^{[\eta,\xi]} \,;\, \tilde{R}, \tilde{S} \rangle & \overset{\text{Reduce}}{\tilde{\longmapsto}} & \langle \tilde{H} \,;\, \tilde{e} \,;\, \tilde{S} \rangle & \text{if } \eta = 0 \text{ and } \tilde{R}[\tilde{v}^{[\eta,\xi]}] \tilde{\mapsto} \tilde{e}
\end{array}
$$

Figure 2.3: Abstract machine transition rules for annotated terms

Configurations in the abstract machine now take the form $\langle \tilde{H} \,;\, \tilde{e} \,;\, \tilde{S} \rangle$, where $\tilde{H}$ is a heap of annotated bindings and $\tilde{S}$ is an annotated abstract machine stack:

$$
\begin{array}{lrl}
\text{Configurations} & \tilde{C} & ::= \quad \langle \tilde{H} \,;\, \tilde{e} \,;\, \tilde{S} \rangle \\
\text{Heaps} & \tilde{H} & ::= \quad \epsilon \mid \tilde{H}, \tilde{b} \\
\text{Stacks} & \tilde{S} & ::= \quad \epsilon \mid \tilde{R}, \tilde{S} \mid \#x, \tilde{S} \\
\text{Reduction contexts} & \tilde{R} & ::= \quad [\,]\, x \mid [\,] +^{\iota} \tilde{e} \mid \texttt{add}^{\iota}_{n}\, [\,] \mid \\
 & & \quad\; \texttt{case}\, [\,]\, \texttt{of}\, \tilde{alts}
\end{array}
$$

The transition rules of the abstract machine are given in figure 2.3. The rules Let and Unwind remain unchanged (adding only a $\tilde{}$ everywhere). However, there are now two Var rules. The rule

$$\langle \tilde{H}, x =^{!} \tilde{e} \,;\, x \,;\, \tilde{S} \rangle \overset{\text{Var-!}}{\tilde{\longmapsto}} \langle \tilde{H} \,;\, \tilde{e} \,;\, \#x, \tilde{S} \rangle$$

takes care of variables bound to closures that shall be updated. It looks up the binding in the heap, removes it, pushes an update marker and evaluates the expression just as the rule Var. The rule

$$\langle \tilde{H}, x =^{\checkmark} \tilde{e} \,;\, x \,;\, \tilde{S} \rangle \overset{\text{Var-}\checkmark}{\tilde{\longmapsto}} \langle \tilde{H} \,;\, \tilde{e} \,;\, \tilde{S} \rangle$$

takes care of variables bound to closures that shall not be updated. It looks up the binding in the heap, removes it and evaluates the expression *without* pushing an update marker. Indeed, this means that the binding will not be updated. Note that we require configurations to be closed so the rule does not apply unless the configuration remains closed. This is important since an open configuration would correspond to dangling pointers in an implementation. If the rule does not apply the computation will go wrong and we will consider the configuration to be ill-annotated. An example of this would be the configuration

$$\langle x =^{\checkmark} y, y =^{\checkmark} 1^{[0,0]} \,;\, y \,;\, [\,] +^{[0,0]} x, \epsilon \rangle$$

which cannot reduce further since there is a reference to y in the binding for x (which is not dead since there is a reference to x on the stack). Note that the transition rules are defined up to garbage equivalence. That is we may, and it is sometimes necessary to, garbage convert the configuration before a transition rule can be applied. Consider for example, the modification of the previous example so that there is no reference to x on the stack.

$$\langle x =^{\checkmark} y, y =^{\checkmark} 1^{[0,0]} \,;\, y \,;\, \epsilon\rangle \equiv \langle y =^{\checkmark} 1^{[0,0]} \,;\, y \,;\, \epsilon\rangle \overset{\text{Var-}\checkmark}{\tilde{\longmapsto}} \langle \epsilon \,;\, 1^{[0,0]} \,;\, \epsilon\rangle$$

We cannot directly apply the transition rule since there is a reference to y in the binding for x. But since the binding for x is dead we may remove it and we can then apply the transition rule. The rule

$$\langle \tilde{H} \,;\, \tilde{v}^{[\eta,\xi]} \,;\, \#x, \tilde{S}\rangle \overset{\text{Update}}{\longmapsto} \langle \tilde{H}, x =^{!} \tilde{v}^{[\eta,\xi]} \,;\, \tilde{v}^{[\eta-1,\xi-1]} \,;\, \tilde{S}\rangle \quad \text{if } \xi \geq 1$$

has been modified to take the annotation on values into account. For the value to perform an update, the upper bound must be nonzero. This reflects that a value with a zero upper bound requires the top of the stack to contain a reduction context. Once the update has been performed the annotation is decreased to record that an update marker has been taken care of. Here we take $\omega - 1$ to be ω and $0 - 1$ to be 0. If the upper bound is zero the computation will go wrong and we will consider the configuration to be ill-annotated. The rule

$$\langle \tilde{H} \,;\, \tilde{v}^{[\eta,\xi]} \,;\, \tilde{R}, \tilde{S}\rangle \overset{\text{Reduce}}{\longmapsto} \langle \tilde{H} \,;\, \tilde{e} \,;\, \tilde{S}\rangle \quad \text{if } \eta = 0 \text{ and } \tilde{R}[\tilde{v}^{[\eta,\xi]}] \overset{\sim}{\mapsto} \tilde{e}$$

has also been modified to take the annotation on values into account. For the value to take part in a reduction the lower bound must be zero. That is the value must not require an update marker on top of the stack. If the lower bound is nonzero the computation will go wrong and we will consider the configuration to be ill-annotated. Again the rule

$$\langle \tilde{H}, x =^{!} \tilde{v}^{[\eta,\xi]} \,;\, x \,;\, \tilde{S}\rangle \overset{\text{Lookup}}{\longmapsto} \langle \tilde{H}, x =^{!} \tilde{v}^{[\eta,\xi]} \,;\, \tilde{v}^{[\eta-1,\xi-1]} \,;\, \tilde{S}\rangle \quad \text{if } \xi \geq 1$$

can be added to make for a more efficient abstract machine.

Again the machine can terminate for three different reason; the computation terminates successfully with a value, a black hole is detected or the computation goes wrong. However, now the computation can go wrong for a number of reasons:

- A binding is erroneously annotated with a ✓. In this case the terminal configuration has the form $\langle \tilde{H}, x =^{\checkmark} \tilde{e} \,;\, x \,;\, \tilde{S}\rangle$ and there is no $\tilde{H}'$ such that $\langle \tilde{H}', x =^{\checkmark} \tilde{e} \,;\, x \,;\, \tilde{S}\rangle \equiv \langle \tilde{H}, x =^{\checkmark} \tilde{e} \,;\, x \,;\, \tilde{S}\rangle$ and $\langle \tilde{H}' \,;\, e \,;\, \tilde{S}\rangle$ is closed.
- An update marker is on top of the stack, but the annotation on the value erroneously specifies that no update marker needs to be taken care of. Thus, the terminal configuration is of the form $\langle \tilde{H} \,;\, \tilde{v}^{[\eta,0]} \,;\, \#x, \tilde{S}\rangle$.

- A reduction context is on top of the stack but the annotation on the value erroneously specifies that there shall be an update marker on top of the stack. Thus, the terminal configuration is of the form $\langle \tilde{H} \,;\, \tilde{v}^{[\eta+1,\xi]} \,;\, \tilde{R}, \tilde{S} \rangle$.

- The configuration is of the form $\langle \tilde{H} \,;\, \tilde{v}^{[0,\xi]} \,;\, \tilde{R}, \tilde{S} \rangle$ but $\tilde{R}[\tilde{v}^{[0,\xi]}] \not\tilde{\rightarrow}$. This cannot happen if we only consider well typed terms (that is well typed in an ordinary type system).

We define Value$\tilde{}$, Blackhole$\tilde{}$ and Wrong$\tilde{}$ to be the sets of terminal configurations of the different forms. and we will let $\tilde{V}$, $\tilde{B}$ and $\tilde{W}$ range over Value$\tilde{}$, Blackhole$\tilde{}$ and Wrong$\tilde{}$ respectively. We will say that a configuration $\tilde{C}$ is ill-annotated if it goes wrong (ie there exists $\tilde{W}$ such that $\tilde{C} \;\tilde{\longmapsto}^*\; \tilde{W}$). Conversely, we will say that a configuration is well-annotated if it does not go wrong. We will also say that a closed term $\tilde{e}$ is ill-annotated/well-annotated if $\langle \epsilon \,;\, \tilde{e} \,;\, \epsilon \rangle$ is ill-annotated/well-annotated.

A crucial property of our analysis should be that it annotates expressions so that the computation cannot go wrong. Our analysis is phrased as a type system and indeed our soundness result says that well-typed terms are well-annotated. An important implication is that if only well typed terms are considered, the cost associated with detecting whether a computation goes wrong can be avoided. This is particularly important for the rule Var-✓ where checking whether the configuration remains closed or not would be very expensive.

2.4 Implementing the abstract machine

As it stands the abstract machine is not well suited to be implemented directly. The main reason for this is that it uses substitution which is not very efficient. However, Sestoft has shown that from this abstract machine one can derive lower level abstract machines which can be implemented efficiently [Ses97]. For our purposes the higher level abstract machine is well-suited. It models the aspects of lower level machines that we are interested in and the higher level makes it tractable for formal proofs. However there are two important aspects which deserve some attention.

First, some implementations garbage collect update markers if they are dead (ie, if there is no reference to them). Since we try to predict statically how many update markers reside on the stack, our implementation cannot be allowed to garbage collect update markers. The objective for garbage collecting update markers is twofold. One being that garbage collecting an update marker saves the cost of performing an update. This saves a strictly bounded amount of work and it only applies to the dead update markers that happen to be on the stack when a garbage collection is triggered. With our analysis many of the markers that potentially could be garbage collected will also never end up on the stack since the corresponding binding often gets annotated with a ✓. The other reason for wanting to garbage collect update markers is that it also reduces

the size of the stack. Unless arbitrarily many update markers are stacked on top of each other (ie without intervening reduction contexts) the saved space is only (a small) constant factor. Unfortunately arbitrarily many update markers can stack up on top of each other as in the following example (where *index* is the function that picks the n'th element of a list).

$$\begin{array}{l}\texttt{let}\ f = \lambda x.\texttt{let}\ x' = id\ x \\ \qquad\qquad\qquad\qquad\qquad\qquad xs = f\ x' \\ \qquad\qquad\qquad\qquad\qquad\ \texttt{in}\ \texttt{cons}\ x\ xs \\ \qquad\qquad y = 1 + 2 \\ \texttt{in}\ index\ n\ (f\ y)\end{array}$$

In this example the number of update markers that stack up on top of each other is linear in n but all of them can be garbage collected away. Thus in an implementation with garbage collection of update markers this program can run without causing a stack overflow (assuming that running out of stack can trigger a garbage collection) but in an implementation without it cannot. Fortunately this seems to occur very rarely in practice and some compilers do not garbage collect update markers. Anyhow the situation is not entirely satisfactory and we will sketch a possible solution in chapter 8.

Another aspect of the abstract machine is that it duplicates values where some implementations do not. If we, for example, evaluate

$$\begin{array}{l}\texttt{let}\ x = 1 + 2 \\ \qquad y = id\ x \\ \texttt{in}\ y\end{array}$$

we will end up with both x and y bound to 3. Some implementations instead create an indirection from y to x which can later be removed by the garbage collector, thereby saving space. This can only reduce the space used by a constant factor but it can have a quite dramatic effect in practice [RW93]. However in most cases the effect is quite small and some implementations do indeed copy values [PJ92]. It should also be possible to modify the abstract machine and our type system to model an implementation that creates indirections but we have not considered that.

Chapter 3

Type system

In this chapter we present a type system in the form of a type directed translation that annotates expressions. We will take the type system by Mossin, Turner and Wadler as our starting point and we will modify it so that it discovers more unnecessary updates and provides information that allows us to bypass update marker checks.

3.1 A few observations

The semantics of section 2.3 specifies that for a binding $x = e$ to be annotated with a ✓ it is required that when (if ever) the binding is used there is only one occurrence of x in the configuration, namely the one that is dereferenced. Our type system is based on the following idea. If, when a binding $x = e$ is created, x occurs only once in the configuration and x never gets duplicated during the computation then x will occur only once when (if ever) it is dereferenced. Thus, the type system will annotate a binding with a ✓, if the corresponding variable occurs once when the binding is created and it can assure that it never gets duplicated. There is an important exception to the above; when a variable occurs once in both branches of a `case`-expression. Then, since eventually only one branch will be taken, we may consider it as occurring only once. [1]

A variable x may get duplicated during the computation in two ways:

- By the rule

$$\langle H\ ;\ v\ ;\ R, S\rangle \overset{\text{Reduce}}{\longmapsto} \langle H\ ;\ e\ ;\ S\rangle \quad \text{if } R[v] \mapsto e$$

[1] Another way to explain this exception is to turn to a lower level semantics. Had we used an abstract machine with environments (such as the mark 2 machine in [Ses97]) we would not have required that a variable occurs once in the configuration but rather that it occurs in at most one environment. Then `case`-expressions would not had been an exception anymore, since the variables in the branches of a case expression always occur in the same environment (until one of the branches is selected).

since when reducing an application (or a `case`-expression) the variable gets substituted into the body of the abstraction (or the `cons`-branch of the `case`) and if the bound variable occurs more than once in the body then x will get duplicated. This is the reason why the binding of x is annotated with an ! in the following example.

$$\begin{array}{l}\texttt{let}\ x =^{!} 1 + 2\ \texttt{in} \\ (\lambda y.y + y)\ x\end{array}$$

- By the rule

$$\langle H\ ;\ v\ ;\ \#x, S\rangle \overset{\text{Update}}{\longmapsto} \langle H, x = v\ ;\ v\ ;\ S\rangle$$

(or by the rule Lookup which is synthesised from Update) since when an update (or lookup) is performed the value gets duplicated and thus also its free variables. In the example

$$\begin{array}{l}\texttt{let}\ x =^{!} 1 + 2\ \texttt{in} \\ \texttt{let}\ f =^{!} \lambda y.x + y\ \texttt{in} \\ f\ 1 + f\ 2\end{array}$$

the abstraction $\lambda y.x + y$ gets duplicated when f is looked up. Thus, since x is a free variable of the abstraction, x gets duplicated. This is the reason why the binding of x is annotated with an !.

3.2 Type language

The type language is given below and is an extension to an ordinary type language. For simplicity the system is monomorphic and our type language does not contain type variables. In chapter 5 we will extend the type language with type variables to allow for a principal typing property.

Bare types	ρ	$::= \quad \texttt{Int} \mid \sigma \to \tau \mid \texttt{List}\,\sigma\,\kappa\,\iota$
Types	τ	$::= \quad \rho^{\iota}$
Binding types	σ	$::= \quad \tau_{\kappa}$

We let ρ range over bare types, that is types without outermost annotations, which includes integers, function types and lists. We will use bare types to give types to bare values (values without outermost annotations). We will get back and explain the different forms of bare types. However first we need to introduce types and binding types. We let τ range over types. The type of an expression $\tilde{e}$ reflects not only the ordinary type of the expression but also the number of update markers the value of the expression (if it terminates) can take care of. Thus, an expression with type $\rho^{[\eta,\xi]}$ must be able to handle any number of markers between η and ξ. For example, $5^{[1,2]}$ has the type $\texttt{Int}^{[1,2]}$. We let σ range over binding types which we will use to give a type to bindings

and variables (whose types will reflect the type of the bindings they refer to). The type of a binding $x =^{\kappa} \tilde{e}$ naturally reflects the type of the expression $\tilde{e}$ but it also reflects the way it is bound. So for example $x =^{\checkmark} 2^{[0,0]} +^{[0,0]} 3^{[0,0]}$ has the type $x : \mathtt{Int}^{[0,0]}_{\checkmark}$. Note that we include the name of the variable in the type of the binding. If a variable x refers to a binding with the type $x : \sigma$ we will say that x has the type σ. An example which might need some explanation is $x =^{!} 2^{[0,0]} +^{[1,1]} 3^{[0,0]}$ which we assign the type $x : \mathtt{Int}^{[0,0]}_{!}$. This may seem a bit surprising at first glance since the addition is annotated with $[1,1]$ we might expect the type to be $x : \mathtt{Int}^{[1,1]}_{!}$. However this is not the case since when we evaluate x we look up the binding, push an update marker and evaluate $2^{[0,0]} +^{[1,1]} 3^{[0,0]}$. The annotation $[1,1]$ then ensures that the update marker we just pushed can be taken care of. However no additional update markers can be handled. Thus there must not be any marker on top of the stack when we start the evaluation of x and therefore we assign the binding the type $x : \mathtt{Int}^{[0,0]}_{!}$.

Let us return to the different forms of bare types. The bare type $\sigma \to \tau$ denotes functions that when applied to a variable (remember expressions can only be applied to variables) with the binding type σ will yield something of type τ. We will sometimes use $\sigma \to^{\iota} \tau$ as syntactic sugar for $(\sigma \to \tau)^{\iota}$. The bare type $\mathtt{List}\,\sigma\,\kappa\,\iota$ denotes lists whose head (if non-empty) has the binding type σ, and whose tail has the binding type $(\mathtt{List}\,\sigma\,\kappa\,\iota)^{\iota}_{\kappa}$. The fact that the head and the tail are given binding types (rather than just types) reflects that `cons` can be applied to variables only. Thus the types of the head and the tail are actually the types of the bindings referred to from the `cons`-cell. An empty list can naturally be given any list-type.

3.3 A subtyping relation

There is a natural subtype relation for our type language. For example, we say that $5^{[1,2]}$ has the type $\mathtt{Int}^{[1,2]}$ since $5^{[1,2]}$ can take care of one or two update markers. But it should also have the type $\mathtt{Int}^{[1,1]}$ since it can indeed take care of just one update marker. Clearly every term of the type $\mathtt{Int}^{[1,2]}$ could also be considered to have the type $\mathtt{Int}^{[1,1]}$ although it is less informative. Therefore we say that $\mathtt{Int}^{[1,2]}$ is a subtype of $\mathtt{Int}^{[1,1]}$.

When we turn to binding types the situation is slightly subtle and we need to have our application in mind. Remember that the idea behind the type system is that we can annotate a binding $x = \tilde{e}$ with a $\checkmark$ if we can assure that we never duplicate x during the computation. This is achieved by preventing the duplication of variables whose type is of the form $\tau_{\checkmark}$. Clearly a binding of the form $x =^{\checkmark} \tilde{e}$ must not be given a type of the form $x : \tau_{!}$ since it would allow that the variable x, referring to the binding, gets duplicated and this must not happen. For bindings annotated with an ! the situation is different. A variable referring to such a binding may freely be duplicated and possible dereferenced several times. However it is not forced to be duplicated. Although

$$\frac{}{\texttt{Int} \leq \texttt{Int}} \qquad \frac{\sigma' \leq \sigma \qquad \tau \leq \tau'}{\sigma \to \tau \leq \sigma' \to \tau'} \qquad \frac{\sigma \leq \sigma' \qquad \kappa' \leq \kappa \qquad \iota' \leq \iota}{\texttt{List}\,\sigma\,\kappa\,\iota \leq \texttt{List}\,\sigma'\,\kappa'\,\iota'}$$

$$\frac{\tau \leq \tau' \qquad \kappa' \leq \kappa}{\tau_\kappa \leq \tau'_{\kappa'}} \qquad \frac{\rho \leq \rho' \qquad \iota' \leq \iota}{\rho^\iota \leq \rho'^{\iota'}}$$

Figure 3.1: The subtyping relation

less informative, we can therefore safely assign the binding a type of the form $\tau_\checkmark$.

Before we go on and define our subtyping relation we will need orderings on the annotations. First we define $[\eta, \xi] \leq [\eta', \xi']$ iff $\eta' \leq \eta$ and $\xi \leq \xi'$. The ordering can be thought of as modelling the capability of (a value annotated with) the annotation. The smaller the annotation the fewer stack configurations it will be able to handle safely. Thus the completely incapable annotation $[\omega, 0]$ is the smallest annotation. A value annotated with $[\omega, 0]$ will require an infinite number of update markers on the stack but it cannot take care of any of them. In contrast the largest annotation $[0, \omega]$ can take care of an arbitrary number of update markers. It is clearly safe to annotate every value with $[0, \omega]$. However then little would be won since the term would behave exactly as the unannotated terms in the standard semantics. We also define an ordering on $\{\checkmark, !\}$ where $\checkmark < !$. Again the ordering models the capability of the annotations. Using these orderings we finally define the subtyping relation which can be found in figure 3.1.

3.4 Contexts

We use Γ to range over typing contexts.

$$\text{Contexts} \quad \Gamma ::= \quad x_1 : \sigma_1,\ ...,\ x_n : \sigma_n$$

A context associates binding types with variables and consists of a sequence of type associations of the form $x : \sigma$. A context may very well contain several occurrences of the same variable, and the ordering of type associations is irrelevant. We will write Γ_0, Γ_1 to denote the concatenation of Γ_0 and Γ_1. As usual we will use contexts when we give a type to a term with free variables. Thus we will say that $\tilde{e}$ has the type τ in a context Γ if we can give $\tilde{e}$ the type τ assuming that the free variables in $\tilde{e}$ has the types given by Γ. However the context also plays another important rôle; it records the number of times each variable occurs in the term. Thus if x occurs n times in $\tilde{e}$ it also occurs n times in Γ (with one important exception, namely if x occurs in different branches of

a case-expression). This may be a bit surprising at first. Consider for example the term $(\lambda^{[0,0]}y.y +^{[0,0]} y)\ x$ with the free variable x. We will be able to say that this term has the type $\mathtt{Int}^{[0,0]}$ in the context $x : \mathtt{Int}_!^{[0,0]}$. According to the reduction relation the term can reduce to $x +^{[0,0]} x$ so we would expect to be able to give $x +^{[0,0]} x$ the same type in the same context. However this will not be possible since x now occurs twice in the term. Instead we can type the term in the context $x : \mathtt{Int}_!^{[0,0]}, x : \mathtt{Int}_!^{[0,0]}$ where x occurs twice. To be able to state a relation between the contexts before and after a reduction we define a rewrite relation on contexts.

$$\Gamma,\ x : \tau_!\ \overset{\text{Dup}}{\rightarrow}\ \Gamma,\ x : \tau_!,\ x : \tau_! \qquad \Gamma,\ x : \sigma\ \overset{\text{Drop}}{\rightarrow}\ \Gamma$$

We have two rewrite rules. The first says that a type association of the form $x : \tau_!$ may be duplicated. This is suppose to model the duplication of a variable x during the computation. Note that we may not duplicate a type association of the form $x : \tau_\checkmark$. This reflects our intention that a variable that refers to a binding which will not be updated, must not be duplicated. The second rule simply allows us to remove a type association. This corresponds to when a variable is dropped during the computation (for example since it occurred in a branch of a case-expression that was not selected). These rewrite rules will play a rôle similar to the weakening and contraction rules in logic. The restricted duplication (ie that we may only duplicate type associations of the form $x : \tau_!$) corresponds to the restricted form of contraction in linear logic [Gir87].

We will let Δ range over distinct typing contexts.

$$\begin{array}{lll} \text{Distinct contexts} & \Delta ::= & x_1 : \sigma_1,\ ...,\ x_n : \sigma_n \\ & & \text{where } x_i \neq x_j \text{ if } i \neq j \end{array}$$

A distinct context is a context that does not contain the same variable more than once. If (and only if) the distinct contexts Δ_0 and Δ_1 have no variables in common we will write Δ_0, Δ_1 for their concatenation. We will use distinct contexts to give a type to declarations and heaps. For example $x =^\checkmark 1^{[0,0]} +^{[0,0]} 2^{[0,0]}, y =^! 3^{[0,0]} +^{[1,1]} 4^{[0,0]}$ has the type $x : \mathtt{Int}_\checkmark^{[0,0]}, y : \mathtt{Int}_!^{[0,0]}$. The fact that we use distinct contexts rather than contexts reflects that a declaration may not bind the same variable more than once.

To relate the type Δ of a heap to the types of the variables referring to the bindings in the heap we define a relation $\Delta \vdash \Gamma$ (we will say that Δ entails Γ). We will let $\Delta \vdash \Gamma$ iff $\Delta \rightarrow^* \Gamma$. The relation simply says that if $\Delta \vdash \Gamma$ then the number of occurrences of a variable in Γ is consistent with the type of the variable in Δ. For example if $x : \tau_\checkmark$ is a type association in Δ then x occurs at most once in Γ (since we may not duplicate such a type association).

3.5 Typing judgements

The analysis is presented in the form of a type directed translation. There are five forms of typing judgements, one for each syntactic category. Judgements

for bare values take the form $\Gamma \vdash v \rightsquigarrow \tilde{v} : \rho$ and shall be read "In the context Γ, the value v can be annotated as $\tilde{v}$ having bare type ρ". We will refer to v as the source and $\tilde{v}$ as the target of the translation. Similarly judgements for expressions take the form $\Gamma \vdash e \rightsquigarrow \tilde{e} : \tau$. Judgements for alternatives take the form $\Gamma \vdash alts \rightsquigarrow \tilde{alts} : \rho \Rightarrow \tau$. Thus we will assign alternatives a type of the form $\rho \Rightarrow \sigma$ where ρ is the bare type of the bare value used to select a branch and τ is the type of the right hand side of the branches. In our small language ρ will always be of list type. Judgements for bindings take the form $\Gamma \vdash b \rightsquigarrow \tilde{b} : (x : \sigma)$. Note that the type of a binding includes the name of the bound variable, ie the type of a binding is actually a type association. Finally, judgements for declarations take the form $\Gamma \vdash d \rightsquigarrow \tilde{d} : \Delta$. Thus, the type of a declaration is a distinct context containing the types of the bindings in $\tilde{d}$. As discussed in the previous section the context in our judgements as usual keeps track of the types of the free variables of the term but it also records the number of times each variable occurs in the term. It should be noted that when typing a binding the context will contain all variables occurring in the right hand side of the binding possibly including the variable bound by the binding. Similarly when typing a declaration the context will contain all variables occurring in the right hand sides of the declaration possibly including the variables bound by the declaration.

3.6 Typing rules

In this section we will present the typing rules of our type system. We will present the rules one by one. For ease of reference all typing rules have been conveniently collected in a few figures in appendix A.

3.6.1 Typing bare values

The rule

$$\text{Abs}\frac{\Gamma_0, \Gamma_1 \vdash e \rightsquigarrow \tilde{e} : \tau}{\Gamma_0 \vdash \lambda x.e \rightsquigarrow \lambda x.\tilde{e} : \sigma \to \tau} \quad \begin{array}{l} x \notin \mathrm{dom}(\Gamma_0) \\ x : \sigma \to^* \Gamma_1 \end{array}$$

is used to type bare abstractions. The key feature of the rule is that if x occurs more than once in $\tilde{e}$ then the abstraction will be assigned a type of the form $\tau_!' \to \tau$ indicating that a variable will be duplicated if it is passed to the abstraction. This is accomplished by first typing $\tilde{e}$ in a context Γ_0, Γ_1, where $x \notin \mathrm{dom}(\Gamma_0)$. Then, if x occurs more than once in $\tilde{e}$, x will occur more than once in Γ_1. Now the side condition $x : \sigma \to \Gamma_1$ specify that we must be able to rewrite $x : \sigma$ to Γ_1 which clearly involves duplicating $x : \sigma$ (since x occurs more than once in Γ_1) so σ must then be of the form $\tau_!'$. The rule

$$\text{Int}\frac{}{\vdash n \rightsquigarrow n : \texttt{Int}}$$

is used to type integers and is straightforward. The only thing worth noticing is that we must type integers in an empty context since an integer does not contain any free variables. The rule

$$\text{Nil}\frac{}{\vdash \texttt{nil} \rightsquigarrow \texttt{nil} : \texttt{List}\,\sigma\,\kappa\,\iota}$$

is used to type the empty list and should be completely straightforward. The rule

$$\text{Cons}\frac{}{x : \sigma_0,\ y : \sigma_1 \vdash \texttt{cons}\,x\,y \rightsquigarrow \texttt{cons}\,x\,y : \texttt{List}\,\sigma\,\kappa\,\iota} \quad \begin{array}{l}\sigma_0 \leq \sigma \\ \sigma_1 \leq (\texttt{List}\,\sigma\,\kappa\,\iota)^{\iota}_{\kappa}\end{array}$$

used to type cons cells might however require some explanation. Saying that a `cons`-value has the type $\texttt{List}\,\sigma\,\kappa\,\iota$ should mean that (the variable referring to the binding containing) the head of the list has the binding type σ and (the variable referring to the binding containing) the tail has the binding type $(\texttt{List}\,\sigma\,\kappa\,\iota)^{\iota}_{\kappa}$. However since we have a subtyping relation on our types it is a desirable property that if we can derive that a term has a type we should be able to derive (in the same context) that it has any supertype as well. We could obtain this property by having a separate subsumption rule but we have chosen not to. Instead we have added subtyping as a side conditions to a few carefully selected rules so that we can show a subsumption lemma.

3.6.2 Typing expressions

The rule

$$\text{Value}\frac{\Gamma \vdash v \rightsquigarrow \tilde{v} : \rho}{\Gamma \vdash v \rightsquigarrow \tilde{v}^{\iota} : \rho^{[\eta,\xi]}} \quad \begin{array}{l}\text{if } \xi > 0 \text{ then } \Gamma \rightarrow^* \Gamma, \Gamma \\ [\eta, \xi] \leq \iota\end{array}$$

is used to type an annotated value. Saying that an annotated value has the type $\rho^{[\eta,\xi]}$ means that the value has to be able to take care of any number of of update markers between η and ξ. So if $\xi > 0$ then the annotated value must be able to take care of at least one update marker. Taking care of an update marker means updating with the value, thus duplicating any free variables of the value. The purpose of the side condition if $\xi > 0$ then $\Gamma \rightarrow^* \Gamma, \Gamma$ is to ensure that these variables may safely be duplicated. Finally the side condition $[\eta, \xi] \leq \iota$ says that the annotation ι on the value may be more capable than the annotation $[\eta, \xi]$ on the type, but not less. The rule

$$\text{Var}\frac{}{x : \tau_{\kappa} \vdash x \rightsquigarrow x : \tau'} \quad \tau \leq \tau'$$

is straightforward and simply records the fact that x occurs once in the term. The side condition is again there to enable the subsumption lemma. The rule

$$\text{App}\frac{\Gamma \vdash e \rightsquigarrow \tilde{e} : \sigma \rightarrow^{[0,0]} \tau}{\Gamma,\ x : \sigma \vdash e\,x \rightsquigarrow \tilde{e}\,x : \tau}$$

however requires some explanation. Naturally if $\tilde{e}$ has the type $\sigma \to^{[0,0]} \tau$ and x has the type σ then $\tilde{e}\,x$ should be given the type τ. But why does the rule require that the function type is annotated with $[0,0]$? The reason is that when evaluation of $\tilde{e}\,x$ starts, a reduction context of the form $[\,]\,x$ is pushed onto the stack. Thus it is crucial that $\tilde{e}$ does not require any update markers on top of the stack. The rule

$$\text{Plus}\frac{\Gamma_0 \vdash e_0 \rightsquigarrow \tilde{e}_0 : \mathtt{Int}^{[0,0]} \qquad \Gamma_1 \vdash e_1 \rightsquigarrow \tilde{e}_1 : \mathtt{Int}^{[0,0]}}{\Gamma_0,\, \Gamma_1 \vdash e_0 + e_1 \rightsquigarrow \tilde{e}_0 +^{\iota} \tilde{e}_1 : \mathtt{Int}^{\iota'}} \quad \iota' \leq \iota$$

ensures that $\tilde{e}_0$ and $\tilde{e}_1$ do not require any update markers on top of the stack (cf the rule App). Otherwise it is completely straightforward. The rule

$$\text{Add}\frac{\Gamma \vdash e \rightsquigarrow \tilde{e} : \mathtt{Int}^{[0,0]}}{\Gamma \vdash \mathtt{add}_n\, e \rightsquigarrow \mathtt{add}^{\iota}_n\, \tilde{e} : \mathtt{Int}^{\iota'}} \quad \iota' \leq \iota$$

is similar to the rule Plus. The rule

$$\text{Let}\frac{\Gamma_0,\, \Gamma_1 \vdash d \rightsquigarrow \tilde{d} : \Delta \qquad \Gamma_2,\, \Gamma_3 \vdash e \rightsquigarrow \tilde{e} : \tau}{\Gamma_0,\, \Gamma_2 \vdash \mathtt{let}\, d\, \mathtt{in}\, e \rightsquigarrow \mathtt{let}\, \tilde{d}\, \mathtt{in}\, \tilde{e} : \tau} \quad \begin{array}{l} \mathrm{dom}(\tilde{d}) \cap \mathrm{dom}(\Gamma_0, \Gamma_2) = \emptyset \\ \Delta \vdash \Gamma_1,\, \Gamma_3 \end{array}$$

ensures that if a `let`-bound variable occurs more than once or may be duplicated then the type of the binding must be of the form $\tau'_!$. This is accomplished in the following way: First, we type $\tilde{d}$ yielding a distinct context Δ containing the types of the bindings in $\tilde{d}$. We then split the context, in which $\tilde{d}$ was typed, into Γ_0 and Γ_1 such that any occurrence of a variable bound by $\tilde{d}$ ends up in Γ_1. This is ensured by the side condition $\mathrm{dom}(\tilde{d}) \cap \mathrm{dom}(\Gamma_0, \Gamma_2) = \emptyset$. Second, we type $\tilde{e}$ and then split the context into Γ_2 and Γ_3. Analogously, any occurrence of a variable bound by $\tilde{d}$ ends up in Γ_3, also ensured by $\mathrm{dom}(\tilde{d}) \cap \mathrm{dom}(\Gamma_0, \Gamma_2) = \emptyset$. Now if a `let`-bound variable x occurs more than once in $\tilde{d}$ and $\tilde{e}$, then x will also occur more than once in Γ_1, Γ_3 and thus the condition $\Delta \vdash \Gamma_1, \Gamma_3$ will force the type of x to be of the form $\tau'_!$, for some τ'. The rule

$$\text{Case}\frac{\Gamma_0 \vdash e \rightsquigarrow \tilde{e} : \rho^{[0,0]} \qquad \Gamma_1 \vdash \mathit{alts} \rightsquigarrow \tilde{\mathit{alts}} : \rho \Rightarrow \tau}{\Gamma_0,\, \Gamma_1 \vdash \mathtt{case}\, e\, \mathtt{of}\, \mathit{alts} \rightsquigarrow \mathtt{case}\, \tilde{e}\, \mathtt{of}\, \tilde{\mathit{alts}} : \tau}$$

simply ensures that $\tilde{e}$ does not require any update markers on top of the stack (cf the rule App). Otherwise it is completely straightforward.

3.6.3 Typing bindings

The rule

$$\text{Bind-}\checkmark\frac{\Gamma \vdash e \rightsquigarrow \tilde{e} : \tau}{\Gamma \vdash x = e \rightsquigarrow x =^{\checkmark} \tilde{e} : (x : \tau_{\checkmark})}$$

simply assigns a binding annotated with a $\checkmark$ a type of the form $\tau_{\checkmark}$. The rule

$$\text{Bind-}!\frac{\Gamma \vdash e \rightsquigarrow \tilde{e} : \rho^{[\eta+1,\xi+1]}}{\Gamma \vdash x = e \rightsquigarrow x =^{!} \tilde{e} : (x : \rho^{[\eta,\xi]}_{\kappa})}$$

is more interesting. The rule requires that the expression in the binding is able to take care of, and allows it to require, an extra update marker. This reflects the fact that when the binding gets evaluated an extra update marker will be pushed onto the stack. Also a binding annotated with an ! may be given a type of either the form $\tau_!$ or the form $\tau_\checkmark$.

3.6.4 Typing declarations

The rules

$$\text{Decl-}\epsilon\frac{}{\vdash \epsilon \rightsquigarrow \epsilon : \epsilon} \qquad \text{Decl}\frac{\Gamma_0 \vdash d \rightsquigarrow \tilde{d} : \Delta \qquad \Gamma_1 \vdash b \rightsquigarrow \tilde{b} : (x : \sigma)}{\Gamma_0, \Gamma_1 \vdash d,\ b \rightsquigarrow \tilde{d},\ \tilde{b} : (\Delta, x : \sigma)}$$

are straightforward. They simply collect the types of the bindings in the declaration.

3.6.5 Typing alternatives

The rule

$$\text{Alts}\frac{\Gamma_0, \Gamma_1 \vdash e_0 \rightsquigarrow \tilde{e}_0 : \tau \qquad \Gamma_0, \Gamma_2, \Gamma_3 \vdash e_1 \rightsquigarrow \tilde{e}_1 : \tau}{\begin{array}{l}\Gamma_0, \Gamma_1, \Gamma_2 \vdash \{\texttt{nil} \Rightarrow e_0; \texttt{cons}\ x\ y \Rightarrow e_1\} \rightsquigarrow \\ \qquad \{\texttt{nil} \Rightarrow \tilde{e}_0; \texttt{cons}\ x\ y \Rightarrow \tilde{e}_1\} : \texttt{List}\ \sigma\ \kappa\ \iota \Rightarrow \tau\end{array}} \quad \begin{array}{l} x, y \notin \mathrm{dom}(\Gamma_0, \Gamma_2) \\ x : \sigma, \\ y : (\texttt{List}\ \sigma\ \kappa\ \iota)^{\iota}_{\kappa} \end{array} \vdash \Gamma_3$$

contains a subtle treatment of contexts. If a variable occurs once in each branch of the `case`-expression and thus twice in the term it may still occur only once in the context. This is achieved by collecting the variables that occur in both branches in a common context Γ_0, thus effectively counting a variable occurring in both branches as one. Finally, the side conditions take care of the variables bound in the `cons`-pattern. They see to that if x or y occurs several times in e_1 then they must have a type of the form $\tau'_!$. It works in essentially the same way as in the rule for abstractions.

Chapter 4

Soundness

In this chapter we will prove the soundness of our type system. By soundness we mean that every well-typed term is indeed well-annotated (in the sense that they do not go wrong).

4.1 Evaluation preserves typings

We will prove the soundness by showing that evaluation preserves typings in the style popularised by Wright and Felleisen [WF94]. Since our notion of evaluation involves transitions between configurations we need to define what it means for a configuration to be well-typed. That is we will need typing judgements of the form $\vdash C \rightsquigarrow \tilde{C} : \tau$. Note that since we require configurations to be closed there is no need for any context. However it turns out that the definition of such a typing relation and its desired properties are not entirely straightforward. We will try to motivate our definition and the properties we will prove about it by studying the expression

$$\begin{array}{l}\texttt{let}\, x = 1 + 2\, \texttt{in} \\ \texttt{let}\, y = (\lambda z.z)\, x\, \texttt{in} \\ y + y\end{array}$$

taken from the introduction. It can safely be annotated as

$$\begin{array}{l}\texttt{let}\, x =^{\checkmark} 1^0 +^1 2^0\, \texttt{in} \\ \texttt{let}\, y =^{!} (\lambda^0 z.z)\, x\, \texttt{in} \\ y +^0 y\end{array}$$

(where 0 and 1 abbreviates $[0,0]$ and $[1,1]$ respectively) having type $\texttt{Int}^{[0,0]}$. If we execute these expressions in the abstract machines we get the two transition sequences given in figure 4.1 and 4.2 respectively. Note that we have named the configurations in the transition sequences as C_0, C_1, etc and $\tilde{C}_0$, $\tilde{C}_1$, etc respectively.

$$\langle \epsilon \,;\, \texttt{let}\, x = 1 + 2 \,\texttt{in}\, \texttt{let}\, y = (\lambda z.z)\, x \,\texttt{in}\, y + y \,;\, \epsilon \rangle \qquad (C_0)$$

$$\overset{\text{Let}}{\longmapsto} \langle x = 1 + 2 \,;\, \texttt{let}\, y = (\lambda z.z)\, x \,\texttt{in}\, y + y \,;\, \epsilon \rangle \qquad (C_1)$$

$$\overset{\text{Let}}{\longmapsto} \langle x = 1 + 2, y = (\lambda z.z)\, x \,;\, y + y \,;\, \epsilon \rangle \qquad (C_2)$$

$$\overset{\text{Unw.}}{\longmapsto} \langle x = 1 + 2, y = (\lambda z.z)\, x \,;\, y \,;\, [\,] + y \rangle \qquad (C_3)$$

$$\overset{\text{Var}}{\longmapsto} \langle x = 1 + 2 \,;\, (\lambda z.z)\, x \,;\, \#y, [\,] + y \rangle \qquad (C_4)$$

$$\overset{\text{Unw.}}{\longmapsto} \langle x = 1 + 2 \,;\, \lambda z.z \,;\, [\,]\, x, \#y, [\,] + y \rangle \qquad (C_5)$$

$$\overset{\text{Red.}}{\longmapsto} \langle x = 1 + 2 \,;\, x \,;\, \#y, [\,] + y \rangle \qquad (C_6)$$

$$\overset{\text{Var}}{\longmapsto} \langle \epsilon \,;\, 1 + 2 \,;\, \#x, \#y, [\,] + y \rangle \qquad (C_7)$$

$$\overset{\text{Unw.}}{\longmapsto} \langle \epsilon \,;\, 1 \,;\, [\,] + 2, \#x, \#y, [\,] + y \rangle \qquad (C_8)$$

$$\overset{\text{Red.}}{\longmapsto} \langle \epsilon \,;\, \texttt{add}_1\ 2 \,;\, \#x, \#y, [\,] + y \rangle \qquad (C_9)$$

$$\overset{\text{Unw.}}{\longmapsto} \langle \epsilon \,;\, 2 \,;\, \texttt{add}_1\ [\,], \#x, \#y, [\,] + y \rangle \qquad (C_{10})$$

$$\overset{\text{Red.}}{\longmapsto} \langle \epsilon \,;\, 3 \,;\, \#x, \#y, [\,] + y \rangle \qquad (C_{11})$$

$$\overset{\text{Upd.}}{\longmapsto} \langle x = 3 \,;\, 3 \,;\, \#y, [\,] + y \rangle \equiv \langle \epsilon \,;\, 3 \,;\, \#y, [\,] + y \rangle \qquad (C_{12})$$

$$\overset{\text{Upd.}}{\longmapsto} \langle y = 3 \,;\, 3 \,;\, [\,] + y \rangle \qquad (C_{13})$$

$$\overset{\text{Red.}}{\longmapsto} \langle y = 3 \,;\, \texttt{add}_3\ y \,;\, \epsilon \rangle \qquad (C_{14})$$

$$\overset{\text{Unw.}}{\longmapsto} \langle y = 3 \,;\, y \,;\, \texttt{add}_3\ [\,] \rangle \qquad (C_{15})$$

$$\overset{\text{Loo.}}{\longmapsto} \langle y = 3 \,;\, 3 \,;\, \texttt{add}_3\ [\,] \rangle \equiv \langle \epsilon \,;\, 3 \,;\, \texttt{add}_3\ [\,] \rangle \qquad (C_{16})$$

$$\overset{\text{Red.}}{\longmapsto} \langle \epsilon \,;\, 6 \,;\, \epsilon \rangle \qquad (C_{17})$$

Figure 4.1: A transition sequence

$$\langle \epsilon \,;\, \texttt{let}\ x =^{\checkmark} 1^0 +^1 2^0\ \texttt{in}\ \texttt{let}\ y =^{!} (\lambda^0 z.z)\ x\ \texttt{in}\ y +^0 y \,;\, \epsilon \rangle \qquad (\tilde{C}_0)$$

$$\overset{\text{Let}}{\tilde{\longmapsto}} \langle x =^{\checkmark} 1^0 +^1 2^0 \,;\, \texttt{let}\ y =^{!} (\lambda^0 z.z)\ x\ \texttt{in}\ y +^0 y \,;\, \epsilon \rangle \qquad (\tilde{C}_1)$$

$$\overset{\text{Let}}{\tilde{\longmapsto}} \langle x =^{\checkmark} 1^0 +^1 2^0, y =^{!} (\lambda^0 z.z)\ x \,;\, y +^0 y \,;\, \epsilon \rangle \qquad (\tilde{C}_2)$$

$$\overset{\text{Unw.}}{\tilde{\longmapsto}} \langle x =^{\checkmark} 1^0 +^1 2^0, y =^{!} (\lambda^0 z.z)\ x \,;\, y \,;\, [\,] +^0 y \rangle \qquad (\tilde{C}_3)$$

$$\overset{\text{Var-!}}{\tilde{\longmapsto}} \langle x =^{\checkmark} 1^0 +^1 2^0 \,;\, (\lambda^0 z.z)\ x \,;\, \#y, [\,] +^0 y \rangle \qquad (\tilde{C}_4)$$

$$\overset{\text{Unw.}}{\tilde{\longmapsto}} \langle x =^{\checkmark} 1^0 +^1 2^0 \,;\, \lambda^0 z.z \,;\, [\,]\ x, \#y, [\,] +^0 y \rangle \qquad (\tilde{C}_5)$$

$$\overset{\text{Red.}}{\tilde{\longmapsto}} \langle x =^{\checkmark} 1^0 +^1 2^0 \,;\, x \,;\, \#y, [\,] +^0 y \rangle \qquad (\tilde{C}_6)$$

$$\overset{\text{Var-}\checkmark}{\tilde{\longmapsto}} \langle \epsilon \,;\, 1^0 +^1 2^0 \,;\, \#y, [\,] +^0 y \rangle \qquad (\tilde{C}_7)$$

$$\overset{\text{Unw.}}{\tilde{\longmapsto}} \langle \epsilon \,;\, 1^0 \,;\, [\,] +^1 2^0, \#y, [\,] +^0 y \rangle \qquad (\tilde{C}_8)$$

$$\overset{\text{Red.}}{\tilde{\longmapsto}} \langle \epsilon \,;\, \texttt{add}_1^1\ 2^0 \,;\, \#y, [\,] +^0 y \rangle \qquad (\tilde{C}_9)$$

$$\overset{\text{Unw.}}{\tilde{\longmapsto}} \langle \epsilon \,;\, 2^0 \,;\, \texttt{add}_1^1\ [\,], \#y, [\,] +^0 y \rangle \qquad (\tilde{C}_{10})$$

$$\overset{\text{Red.}}{\tilde{\longmapsto}} \langle \epsilon \,;\, 3^1 \,;\, \#y, [\,] +^0 y \rangle \qquad (\tilde{C}_{11})$$

$$\overset{\text{Upd.}}{\tilde{\longmapsto}} \langle y =^{!} 3^1 \,;\, 3^0 \,;\, [\,] +^0 y \rangle \qquad (\tilde{C}_{12})$$

$$\overset{\text{Red.}}{\tilde{\longmapsto}} \langle y =^{!} 3^1 \,;\, \texttt{add}_3^0\ y \,;\, \epsilon \rangle \qquad (\tilde{C}_{13})$$

$$\overset{\text{Unw.}}{\tilde{\longmapsto}} \langle y =^{!} 3^1 \,;\, y \,;\, \texttt{add}_3^0\ [\,] \rangle \qquad (\tilde{C}_{14})$$

$$\overset{\text{Loo.}}{\tilde{\longmapsto}} \langle y =^{!} 3^1 \,;\, 3^0 \,;\, \texttt{add}_3^0\ [\,] \rangle \equiv \langle \epsilon \,;\, 3^0 \,;\, \texttt{add}_3^0\ [\,] \rangle \qquad (\tilde{C}_{15})$$

$$\overset{\text{Red.}}{\tilde{\longmapsto}} \langle \epsilon \,;\, 6^0 \,;\, \epsilon \rangle \qquad (\tilde{C}_{16})$$

Figure 4.2: An annotated transition sequence

There are a number of properties one might be tempted to require from our typing relation for configurations.

1. If $\vdash e \rightsquigarrow \tilde{e} : \tau$ then the corresponding initial configurations should be related as well, that is in our example $\vdash C_0 \rightsquigarrow \tilde{C}_0 : \mathtt{Int}^{[0,0]}$.
2. The relation should be preserved by transitions, that is in our example $\vdash C_1 \rightsquigarrow \tilde{C}_1 : \mathtt{Int}^{[0,0]}$, $\vdash C_2 \rightsquigarrow \tilde{C}_2 : \mathtt{Int}^{[0,0]}$ and so forth.
3. If $\vdash C \rightsquigarrow \tilde{C} : \tau$ then C should be a terminal configuration iff $\tilde{C}$ is a terminal configuration.
4. If $\vdash C \rightsquigarrow \tilde{C} : \tau$ then C and $\tilde{C}$ must not be wrong, ie $C \notin$ Wrong and $\tilde{C} \notin$ Wrong$^{\sim}$.
5. If $\vdash C \rightsquigarrow \tilde{C} : \tau$ then C and $\tilde{C}$ should have the same shape, that is if we remove the annotations from $\tilde{C}$ we should obtain C.

However 1, 2 and 3 are contradictory since they would imply that the two transition sequences in our example would be of equal length, which they are not. The reason is of course that by annotating the binding of x with $\checkmark$ we avoid an update and thus save a transition. This also explains why the configurations from C_7 to C_{11} do not have the same shape (there is an extra update marker for x on the stack) as their annotated counterparts. It should be clear that we must be able to discard update markers when we annotate a configuration thus giving up property 5. We will refer to these update markers as the discarded update markers. We also need to modify property 2 and 3 in order for them to hold. The following two propositions is the result of modifying property 2.

Proposition 4.1.1 (Source transition)
If $\vdash C \rightsquigarrow \tilde{C} : \tau$ and $C \longmapsto C'$ then there exists $\tilde{C}'$ such that

- $\tilde{C} \tilde{\longmapsto}^{0/1} \tilde{C}'$
- $\vdash C' \rightsquigarrow \tilde{C}' : \tau$.

Proposition 4.1.2 (Target transition)
If $\vdash C \rightsquigarrow \tilde{C} : \tau$ and $\tilde{C} \tilde{\longmapsto} \tilde{C}'$ then there exists C' such that

- $C \longmapsto^{+} C'$
- $\vdash C' \rightsquigarrow \tilde{C}' : \tau$.

The source transition proposition states that if the source of the translation (the configuration without annotations) can evaluate one step then the target of the translation (the annotated configuration) can match that by taking a step or by taking no step at all. The case where no step is needed is of course when the source takes care of a discarded update marker that does not exist in the

target. In our example this happens in the transition $C_{11} \longmapsto C_{12}$. Indeed our typing rules for configurations will allow that both $\vdash C_{11} \rightsquigarrow \tilde{C}_{11} : \mathtt{Int}^{[0,0]}$ and $\vdash C_{12} \rightsquigarrow \tilde{C}_{11} : \mathtt{Int}^{[0,0]}$. The target transition proposition states that if the target of the translation can evaluate one step then the source of the translation can match that by taking one or more steps. The source of course needs to take more than one step if it has to take care of a discarded update marker that does not exist in the target. In our example, to match $\tilde{C}_{11} \tilde{\longmapsto} \tilde{C}_{12}$ the source needs to take two steps: $C_{11} \longmapsto C_{12}$ and $C_{12} \longmapsto C_{13}$. Finally the following lemma is the result of modifying property 3 and then combining it with property 4.

Lemma 4.1.3 (Terminal configuration property)
If $\vdash C \rightsquigarrow \tilde{C} : \tau$ then

(i) If $C \in \text{Value}$ then $\tilde{C} \in \text{Value}^{\tilde{}}$

(ii) If $\tilde{C} \in \text{Value}^{\tilde{}}$ then $C \longmapsto^* V$ and $\vdash V \rightsquigarrow \tilde{C} : \tau$

(iii) $C \in \text{Blackhole}$ iff $\tilde{C} \in \text{Blackhole}^{\tilde{}}$

(iv) $C \notin \text{Wrong}$ and $\tilde{C} \notin \text{Wrong}^{\tilde{}}$

Again, the update markers that only exist in the source show up. This time in clause (ii): if the target is a value then the source is not necessarily a value since it might be necessary to take care of some update markers that do not exist in the target.

Before we go on and prove the source and target transition properties as well as the terminal configuration property we naturally need to define the typing relation for configurations. But we also need to establish a whole range of properties of our type system. We therefore postpone those proofs to section 4.4. The outline of the rest of this chapter is as follows. We start by defining the typing relation in section 4.2. Then in section 4.3 we go on and prove the soundness of the type system using the properties stated in this section. Finally in section 4.4 we establish a whole range of properties we need to complete the postponed proofs.

4.2 Typing configurations

We will now go on and define our typing relation on configurations. However, to do that we need to define typing relations for the different components of the configurations. Since we can consider a heap as a declaration there is no need for a separate typing relation for heaps. Consequently we will write $\Gamma \vdash H \rightsquigarrow \tilde{H} : \Delta$ when H (considered as a declaration) can be annotated as $\tilde{H}$ (considered as a declaration). To define the typing relation for stacks we first need to define a typing relation for reduction contexts. The typing judgements for reduction contexts take the form $\Gamma \vdash R \rightsquigarrow \tilde{R} : [\rho^{[0,0]}]\tau$ and means that if a term $\tilde{e}$ of type

$\rho^{[0,0]}$ in context Γ' is plugged into $\tilde{R}$ then the result $\tilde{R}[\tilde{e}]$ has type τ in context Γ, Γ'. The typing rules for reduction contexts are derived from the rules for expressions of the corresponding form. For example from the rule

$$\text{App}\frac{\Gamma \vdash e \rightsquigarrow \tilde{e} : \sigma \to^{[0,0]} \tau}{\Gamma,\ x : \sigma \vdash e\ x \rightsquigarrow \tilde{e}\ x : \tau}$$

for applications we can derive the rule

$$\text{AppR}\frac{}{x : \sigma \vdash [\,]\ x \rightsquigarrow [\,]\ x : [\sigma \to^{[0,0]} \tau]\tau}$$

for applicative contexts. The rest of the typing rules are derived in the same way and are given in figure A.6 in appendix A. Our typing judgements for stacks take the form $\Gamma \vdash S \rightsquigarrow \tilde{S} : \Delta\ ;\ [\tau_0]\tau_1$. where Δ corresponds to the types of the update markers in the stack. Since we think of them as binders they are given a type of the same form as given to a heap. The type τ_0 is the type the stack requires the expression in the configuration to have. If so, τ_1 will be the type of the whole configuration. The rule

$$\text{Stack-}\epsilon\frac{}{\vdash \epsilon \rightsquigarrow \epsilon : \epsilon\ ;\ [\tau]\tau}$$

simply states that, for the empty stack, the expression in the configuration may have any type and that will be the type of the whole configuration. The rule

$$\text{Stack-}R\frac{\Gamma_0 \vdash R \rightsquigarrow \tilde{R} : [\rho^{[0,0]}]\tau_0 \qquad \Gamma_1 \vdash S \rightsquigarrow \tilde{S} : \Delta\ ;\ [\tau_0]\tau_1}{\Gamma_0,\ \Gamma_1 \vdash R, S \rightsquigarrow \tilde{R}, \tilde{S} : \Delta\ ;\ [\rho^{[0,0]}]\tau_1}$$

handles the case when a reduction context is on top of the stack. The key feature of the rule is that it requires the expression to have a type of the form $\rho^{[0,0]}$. Thus the expression need not be able to care of and it must not require any update markers on top of the stack. The rule

$$\text{Stack-}\#\frac{\Gamma \vdash S \rightsquigarrow \tilde{S} : \Delta\ ;\ [\tau_0]\tau_1}{\Gamma \vdash \#x,\ S \rightsquigarrow \#x,\ \tilde{S} : (\Delta,\ x : \rho_\kappa^{[\eta,\xi]}\ ;\ [\rho^{[\eta+1,\xi+1]}]\tau_1)} \quad \rho^{[\eta,\xi]} \leq \tau_0$$

handles the case when the stack is of the form $\#x, \tilde{S}$. If the stack $\tilde{S}$ requires an expression of type τ_0 and $\rho^{[\eta,\xi]}$ is a subtype of τ_0 then the stack $\#x, \tilde{S}$ will require an expression of type $\rho^{[\eta+1,\xi+1]}$, that is an expression that can take care of, and is allowed to require, the extra update marker. When x eventually gets updated, it will be updated with a value $\tilde{v}^\iota$ of type $\rho^{[\eta+1,\xi+1]}$. This will create a binding of the form $x =^! \tilde{v}^\iota$ which can be given a type of the form $x : \rho_\kappa^{[\eta,\xi]}$ for any κ. Thus we extend Δ with $x : \rho_\kappa^{[\eta,\xi]}$. Finally the rule

$$\text{Stack-}\#\text{-discard}\frac{\Gamma \vdash S \rightsquigarrow \tilde{S} : \Delta\ ;\ [\tau_0]\tau_1}{\Gamma \vdash \#x,\ S \rightsquigarrow \tilde{S} : \Delta\ ;\ [\tau_2]\tau_1} \quad \tau_2 \leq \tau_0$$

allows update markers to be discarded. Note that when a marker is discarded Δ is not extended and the typing rule for configurations will prevent us from erroneously discarding update markers. Finally typing judgements for configurations take the form $\vdash C \rightsquigarrow \tilde{C} : \tau$ where τ simply is the type of the result of evaluating the configuration. Note that since we only consider closed configurations there is no need for any context. The rule for typing configurations is as follows.

$$\text{Config}\frac{\begin{array}{l}\Gamma_0 \vdash H \rightsquigarrow \tilde{H} : \Delta_0 \\ \Gamma_1 \vdash e \rightsquigarrow \tilde{e} : \tau_0 \\ \Gamma_2 \vdash S \rightsquigarrow \tilde{S} : \Delta_1 \,;\, [\tau_0]\tau_1\end{array}}{\vdash \langle H \,;\, e \,;\, S\rangle \rightsquigarrow \langle \tilde{H} \,;\, \tilde{e} \,;\, \tilde{S}\rangle : \tau_1} \quad \Delta_0,\, \Delta_1 \vdash \Gamma_0,\, \Gamma_1,\, \Gamma_2$$

The rule ensures that if a variable occurs several times in the configuration or may be duplicated, then the corresponding binding in the heap is annotated with an !. This is achieved as follows: First, we type $\tilde{H}$ in a context Γ_0 yielding a distinct context Δ_0 containing the types of the bindings in $\tilde{H}$. Second, we type $\tilde{e}$ in a context Γ_1 yielding some type τ_0. Third, we type $\tilde{S}$ in a context Γ_2 yielding a distinct context Δ_1, corresponding to the types of the update markers, and a type τ_1 giving the type of the whole configuration. Now, if a variable x occurs more than once in $\tilde{H}$, $\tilde{e}$ and $\tilde{S}$ then x will also occur more than once in $\Gamma_0, \Gamma_1, \Gamma_2$ and thus the condition $\Delta_0, \Delta_1 \vdash \Gamma_0, \Gamma_1, \Gamma_2$ will force the type of x to be of the form $\tau_!$, for some τ. It should be pointed out that the typing relation for configurations is defined up to garbage equivalence. That is we are allowed (and sometimes required) to garbage convert the configuration before we can apply the typing rule.

4.3 Soundness theorem

We are now ready to state and prove our soundness theorem.

Theorem 4.3.1 (Soundness)
If $\vdash e \rightsquigarrow \tilde{e} : \tau$ then

(i) $\langle \epsilon \,;\, e \,;\, \epsilon\rangle \;\vdash\!\!\rightarrow^* V$ iff $\langle \epsilon \,;\, \tilde{e} \,;\, \epsilon\rangle \vdash \tilde{\rightarrow}^* \tilde{V}$.

(ii) $\langle \epsilon \,;\, e \,;\, \epsilon\rangle \;\vdash\!\!\rightarrow^* B$ iff $\langle \epsilon \,;\, \tilde{e} \,;\, \epsilon\rangle \vdash \tilde{\rightarrow}^* \tilde{B}$.

(iii) $\langle \epsilon \,;\, e \,;\, \epsilon\rangle$ diverges iff $\langle \epsilon \,;\, \tilde{e} \,;\, \epsilon\rangle$ diverges

(iv) $\langle \epsilon \,;\, e \,;\, \epsilon\rangle \;\not\vdash\!\!\rightarrow^* W$ and $\langle \epsilon \,;\, \tilde{e} \,;\, \epsilon\rangle \not\vdash \tilde{\rightarrow}^* \tilde{W}$.

Proof 4.3.2 (Soundness)
The soundness theorem follows easily from source and target transition and the terminal configuration property. We will only prove (i). The others follow analogously. Assume

$$\vdash e \rightsquigarrow \tilde{e} : \tau.$$

By definition

$$\vdash \langle \epsilon \,;\, e \,;\, \epsilon \rangle \rightsquigarrow \langle \epsilon \,;\, \tilde{e} \,;\, \epsilon \rangle : \tau.$$

We will first consider the left to right implication. Thus assume that

$$\langle \epsilon \,;\, e \,;\, \epsilon \rangle \;\longmapsto^*\; V.$$

Now by the source transition property we know that there exists a corresponding annotated transition sequence

$$\langle \epsilon \,;\, \tilde{e} \,;\, \epsilon \rangle \tilde{\longmapsto}^* \tilde{C}$$

such that

$$\vdash V \rightsquigarrow \tilde{C} : \tau.$$

Finally by the terminal configuration property we know that

$$\tilde{C} \in \text{Value}^{\sim}$$

as required. Now let us turn to the right to left implication and thus assume that

$$\langle \epsilon \,;\, \tilde{e} \,;\, \epsilon \rangle \tilde{\longmapsto}^* \tilde{V}.$$

By the target transition property we can construct a corresponding transition sequence

$$\langle \epsilon \,;\, e \,;\, \epsilon \rangle \;\longmapsto^*\; C$$

where

$$\vdash C \rightsquigarrow \tilde{V} : \tau.$$

Since there may be discarded update markers on C's stack, C is not necessarily a value configuration. However by the terminal configuration property we know that these update markers can be taken care of, that is there exists a transition sequence

$$C \;\longmapsto^*\; V.$$

Thus from $\langle \epsilon \,;\, e \,;\, \epsilon \rangle \;\longmapsto^*\; C$ and $C \;\longmapsto^*\; V$ we may conclude that

$$\langle \epsilon \,;\, e \,;\, \epsilon \rangle \;\longmapsto^*\; V$$

as required.

An important consequence of the fact that well-typed terms cannot go wrong is that if we restrict ourselves to well-typed terms the implementation need not check for whether a computation goes wrong or not. This is important since in our annotated language these checks would be expensive.

4.4 Postponed proofs

In this section we will give the postponed proofs of source and target transition and the terminal configuration lemma. However, before we can do so we need to establish a whole range of properties of the building blocks of our type system. We will regularly omit the details of proofs where they follow by a routine induction.

4.4.1 Context rewriting and context entailment

The context rewriting relation and context entailment relation play a central rôle in our type system. Consequently our proofs heavily uses a number of properties of context rewriting and context entailment. We will need the following properties of context rewriting.

Lemma 4.4.1 (Context rewrite lemma)

(i) If $\Gamma \rightarrow^* \Gamma'$ then $\Gamma, \Gamma'' \rightarrow^* \Gamma', \Gamma''$.

(ii) If $\Gamma \rightarrow^* \Gamma'$ then $\Gamma[\vec{x}{:=}\vec{y}] \rightarrow^* \Gamma'[\vec{x}{:=}\vec{y}]$.

(iii) If $\Gamma_0 \rightarrow^* \Gamma_1$ and $\Gamma_0' \leq \Gamma_0$ then there exist Γ_1' such that $\Gamma_0' \rightarrow^* \Gamma_1'$ and $\Gamma_1' \leq \Gamma_1$.

(iv) If $\Gamma \rightarrow^* \Gamma'$ then $\mathrm{dom}(\Gamma) \supseteq \mathrm{dom}(\Gamma')$.

Proof 4.4.2
Each of the properties are proved by induction over the length of the rewriting sequence.

We will also need a number of properties of context entailment.

Lemma 4.4.3 (Context entailment lemma)

(i) If $\Delta \vdash \Gamma$ and $\Gamma \rightarrow^* \Gamma'$ then $\Delta \vdash \Gamma'$.

(ii) If $\Delta \vdash \Gamma$ then $\mathrm{dom}(\Delta) \supseteq \mathrm{dom}(\Gamma)$

(iii) If $\Delta_0 \vdash \Gamma_0$, $\Delta_1 \vdash \Gamma_1$ and $\mathrm{dom}(\Delta_0) \cap \mathrm{dom}(\Delta_1) = \emptyset$, then $\Delta_0, \Delta_1 \vdash \Gamma_0, \Gamma_1$.

(iv) If $\Delta, x : \sigma \vdash \Gamma, x : \sigma'$ then $\sigma \equiv \sigma'$.

(v) If $\Delta, x : \tau_\checkmark \vdash \Gamma, x : \tau_\checkmark$ then $\Delta \vdash \Gamma$

Proof 4.4.4
The first three results follows immediately from the context rewrite lemma. The two last ones are proved by induction over the rewriting sequence (remember that $\Delta \vdash \Gamma$ iff $\Delta \rightarrow^* \Gamma$) and relies on the fact we only allow a distinct context to the left of the turnstyle.

In the proofs of the rest of this section we will often use the properties of context rewriting and context entailment without an explicit reference to the context rewriting lemma and the context entailment lemma.

4.4.2 Free and bound variables lemmas

A key property of our type system is that a variable occurs in the context if and only if it occurs in the corresponding term. This fact is expressed by the following lemma.

Lemma 4.4.5 (Free variables lemma)

(i) If $\Gamma \vdash e \rightsquigarrow \tilde{e} : \tau$ then $\mathrm{fv}(e) = \mathrm{dom}(\Gamma) = \mathrm{fv}(\tilde{e})$.

(ii) If $\Gamma \vdash H \rightsquigarrow \tilde{H} : \Delta$ then $\mathrm{fv}(H) = \mathrm{dom}(\Gamma) = \mathrm{fv}(\tilde{H})$.

(iii) If $\Gamma \vdash S \rightsquigarrow \tilde{S} : \Delta \,;\, [\tau_0]\tau_1$ then $\mathrm{fv}(S) = \mathrm{dom}(\Gamma) = \mathrm{fv}(\tilde{S})$.

Proof 4.4.6
We first prove (i) by induction over the height of the derivation of $\Gamma \vdash e \rightsquigarrow \tilde{e} : \tau$. We can then prove (ii) and (iii) by induction over the size of H and S respectively.

We also need the following lemma which relates the variables bound in a heap (or stack) to the variables that show up in the distinct context we give as the type to the heap (or stack).

Lemma 4.4.7 (Bound variables lemma)

(i) If $\Gamma \vdash H \rightsquigarrow \tilde{H} : \Delta$ then $\mathrm{dom}(H) = \mathrm{dom}(\Delta) = \mathrm{dom}(\tilde{H})$.

(ii) If $\Gamma \vdash S \rightsquigarrow \tilde{S} : \Delta \,;\, [\tau_0]\tau_1$ then $\mathrm{dom}(S) \supseteq \mathrm{dom}(\Delta) = \mathrm{dom}(\tilde{S})$.

Proof 4.4.8
By induction over the size of H and S respectively.

4.4.3 Subsumption lemma

In any type system with a subtyping relation one would expect that if e can be given the type τ and τ is a subtype of τ' then e can be given the type τ'. This is sometimes assured by having a subsumption rule which allows subtyping to be applied. We have chosen to instead build subtyping into the leaf rules so that we can show the following subsumption lemma. It turns out that by not having a separate subsumption rule we retain a syntax-directed system and our proofs can be liberated from a whole lot of clutter.

Lemma 4.4.9 (Subsumption lemma)
If $\Gamma \vdash e \rightsquigarrow \tilde{e} : \tau$, $\Gamma' \leq \Gamma$ and $\tau \leq \tau'$ and then $\Gamma' \vdash e \rightsquigarrow \tilde{e} : \tau'$.

Proof 4.4.10
By induction over the derivation of $\Gamma \vdash e \rightsquigarrow \tilde{e} : \tau$.

4.4.4 Substitution lemma

As one can expect our type system enjoys a substitution property.

Lemma 4.4.11 (Substitution lemma)
If $\Gamma \vdash e \rightsquigarrow \tilde{e} : \tau$ then $\Gamma[\vec{x}:=\vec{y}] \vdash e[\vec{x}:=\vec{y}] \rightsquigarrow \tilde{e}[\vec{x}:=\vec{y}] : \tau$

Proof 4.4.12
By induction over the derivation of $\Gamma \vdash e \rightsquigarrow \tilde{e} : \tau$.

It should be noted that we only allow variables to be substituted for variables not for terms. Due to the restricted syntax substituting a variable for a term could yield an ill-formed term. Besides, the semantics only substitutes variables for variables so there is no need for such a substitution.

4.4.5 Unwind lemma

The following lemma states that if we can type an expression of the form $\tilde{R}[\tilde{e}]$ then we can type $\tilde{R}$ and $\tilde{e}$ as well. We will later use this lemma when we prove that typings are preserved by the transition Unwind.

Lemma 4.4.13 (Unwind lemma)
If $\Gamma \vdash R[e] \rightsquigarrow \tilde{R}[\tilde{e}] : \tau$ then there exist Γ_0, Γ_1 and ρ such that

- $\Gamma_0 \vdash R \rightsquigarrow \tilde{R} : [\rho^{[0,0]}]\tau$
- $\Gamma_1 \vdash e \rightsquigarrow \tilde{e} : \rho^{[0,0]}$
- $\Gamma \equiv \Gamma_0, \Gamma_1$

Proof 4.4.14
Since the typing rules for reduction contexts are derived from the corresponding rules for expressions it is a simple matter to prove the lemma by case analysis on R and inspection of the typing rules.

4.4.6 Discarded update markers lemma

So far the argued properties of the type system have been rather straightforward or standard. However, the discarded update markers lemma is right at the heart of the type system. Let us first recapitulate the rule

$$\text{Stack-\#-discard}\frac{\Gamma \vdash S \rightsquigarrow \tilde{S} : \Delta\,;[\tau_0]\tau_1}{\Gamma \vdash \#x,\ S \rightsquigarrow \tilde{S} : \Delta\,;[\tau_2]\tau_1} \quad \tau_2 \leq \tau_0$$

which allows update markers to be discarded by the type directed translation. This means that the typing rules for configurations are not syntax directed and that the source and target of the translation might be of different shapes. The significance of the discarded update markers lemma is that is says that these discarded update markers are in fact redundant. That is that a binding created by the update signified by such an update marker can immediately be removed by garbage conversion. Another way to put it is that when the second component of the configuration is a value we may take care of any discarded update marker that happens to be on top of the stack without adding any live binding to the heap. And we can repeat this process until the top of the stack does not contain any discarded update markers. This is expressed in the following lemma.

Lemma 4.4.15 (Discarded update markers lemma)
If $\vdash \langle H\ ;\ v\ ;\ S_0\rangle \rightsquigarrow \tilde{C} : \tau$ then there exist $n \geq 0$ and S_i for $1 \leq i \leq n$ such that

- $\langle H\ ;\ v\ ;\ S_0\rangle \overset{\text{Update}}{\longmapsto} \langle H\ ;\ v\ ;\ S_1\rangle \overset{\text{Update}}{\longmapsto} \ldots \overset{\text{Update}}{\longmapsto} \langle H\ ;\ v\ ;\ S_n\rangle$
- $\vdash \langle H\ ;\ v\ ;\ S_i\rangle \rightsquigarrow \tilde{C} : \tau$
- If $S_n \equiv \epsilon$ then $\tilde{C} \equiv \langle \tilde{H}\ ;\ \tilde{v}^{\iota}\ ;\ \epsilon\rangle$.
- If $S_n \equiv R, S$ then $\tilde{C} \equiv \langle \tilde{H}\ ;\ \tilde{v}^{\iota}\ ;\ \tilde{R}, \tilde{S}\rangle$.
- If $S_n \equiv \#x, S$ then $\tilde{C} \equiv \langle \tilde{H}\ ;\ \tilde{v}^{\iota}\ ;\ \#x, \tilde{S}\rangle$.

Proof 4.4.16 (Discarded update markers lemma)
Assume

$$\vdash \langle H\ ;\ v\ ;\ S_0\rangle \rightsquigarrow \tilde{C} : \tau.$$

We proceed by induction over the size of S_0. By inspection of the typing rules we see that the derivation of $\vdash \langle H\ ;\ v\ ;\ S_0\rangle \rightsquigarrow \tilde{C} : \tau$ is of the following form.

$$\text{Config}\frac{\begin{array}{l}\Gamma_0 \vdash H \rightsquigarrow \tilde{H} : \Delta_0 \\ \Gamma_1 \vdash v \rightsquigarrow \tilde{v}^{\iota} : \tau_0 \\ \Gamma_2 \vdash S_0 \rightsquigarrow \tilde{S} : \Delta_1\ ;\ [\tau_0]\tau\end{array}}{\vdash \langle H\ ;\ v\ ;\ S_0\rangle \rightsquigarrow \langle \tilde{H}\ ;\ \tilde{v}^{\iota}\ ;\ \tilde{S}\rangle : \tau} \quad \Delta_0, \Delta_1 \vdash \Gamma_0, \Gamma_1, \Gamma_2$$

Now the outermost rule used in the derivation of $\Gamma_2 \vdash S_0 \rightsquigarrow \tilde{S} : \Delta_1\ ;\ [\tau_0]\tau$ can be either Stack-ϵ, Stack-R, Stack-# or Stack-#-discard. In the three first cases the result follows immediately by taking $n = 0$. Consider therefore the case where Stack-#-discard was used. That is when the derivation of $\Gamma_2 \vdash S_0 \rightsquigarrow \tilde{S} : \Delta_1\ ;\ [\tau_0]\tau$ is of the following form.

$$\text{Stack-\#-discard}\frac{\Gamma_2 \vdash S_1 \rightsquigarrow \tilde{S} : \Delta_1\ ;\ [\tau_1]\tau}{\Gamma_2 \vdash \#x, S_1 \rightsquigarrow \tilde{S} : \Delta_1\ ;\ [\tau_0]\tau} \quad \tau_0 \leq \tau_1$$

By definition

$$\langle H\,;\,v\,;\,\#x, S_1\rangle \overset{\text{Update}}{\longmapsto} \langle H, x = v\,;\,v\,;\,S_1\rangle$$

but we need to show that $\langle H; v; \#x, S_1\rangle \overset{\text{Update}}{\longmapsto} \langle H; v; S_1\rangle$ which holds if we can show that the binding of x is dead (since the transition rules are defined up to garbage equivalence). This amounts to showing that $x \notin \mathrm{fv}(H) \cup \mathrm{fv}(v) \cup \mathrm{fv}(S_1)$. We first note that

$$x \notin \mathrm{dom}(H) \cup \mathrm{dom}(S_1)$$

since otherwise $\langle H\,;\,v\,;\,\#x, S_1\rangle$ would be ill-formed (since x would occur bound twice: either once in the heap and once in the stack or twice in the stack). We know that Δ_0 and Δ_1 are the types of the bindings in H and the markers in S_0 so by the bound variables lemma

$$x \notin \mathrm{dom}(\Delta_0) \cup \mathrm{dom}(\Delta_1)$$

and since $\Delta_0, \Delta_1 \vdash \Gamma_0, \Gamma_1, \Gamma_2$

$$x \notin \mathrm{dom}(\Gamma_0) \cup \mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2).$$

But we know that H, e and S can be typed in Γ_0, Γ_1 and Γ_2 respectively. Thus

$$x \notin \mathrm{fv}(H) \cup \mathrm{fv}(v) \cup \mathrm{fv}(S_1)$$

and we may conclude that

$$\langle H\,;\,v\,;\,\#x, S_1\rangle \overset{\text{Update}}{\longmapsto} \langle H\,;\,v\,;\,S_1\rangle.$$

If we can show that $\vdash \langle H; v; S_1\rangle \rightsquigarrow \tilde{C} : \tau$ we may apply the induction hypothesis. We start by using the subsumption lemma and $\tau_0 \leq \tau_1$ to yield that

$$\Gamma_1 \vdash v \rightsquigarrow \tilde{v}^{\iota} : \tau_1.$$

We can then derive

$$\vdash \langle H\,;\,v\,;\,S_1\rangle \rightsquigarrow \tilde{C} : \tau$$

as follows.

$$\text{Config}\frac{\begin{array}{l}\Gamma_0 \vdash H \rightsquigarrow \tilde{H} : \Delta_0 \\ \Gamma_1 \vdash v \rightsquigarrow \tilde{v}^{\iota} : \tau_1 \\ \Gamma_2 \vdash S_1 \rightsquigarrow \tilde{S} : \Delta_1\,;\,[\tau_1]\tau\end{array}}{\vdash \langle H\,;\,v\,;\,S_1\rangle \rightsquigarrow \langle \tilde{H}\,;\,\tilde{v}^{\iota}\,;\,\tilde{S}\rangle : \tau} \quad \Delta_0, \Delta_1 \vdash \Gamma_0, \Gamma_1, \Gamma_2$$

Now by using the induction hypothesis we know that there exist $n \geq 1$ and S_i for $2 \leq i \leq n$ such that

- $\langle H\,;\,v\,;\,S_1\rangle \overset{\text{Update}}{\longmapsto} \langle H\,;\,v\,;\,S_2\rangle \overset{\text{Update}}{\longmapsto} \ldots \overset{\text{Update}}{\longmapsto} \langle H\,;\,v\,;\,S_n\rangle$

- $\vdash \langle H\ ;\ v\ ;\ S_i \rangle \rightsquigarrow \tilde{C} : \tau$
- If $S_n \equiv \epsilon$ then $\tilde{C} \equiv \langle \tilde{H}\ ;\ \tilde{v}^\iota\ ;\ \epsilon \rangle$.
- If $S_n \equiv R, S$ then $\tilde{C} \equiv \langle \tilde{H}\ ;\ \tilde{v}^\iota\ ;\ \tilde{R}, \tilde{S} \rangle$.
- If $S_n \equiv \#x, S$ then $\tilde{C} \equiv \langle \tilde{H}\ ;\ \tilde{v}^\iota\ ;\ \#x, \tilde{S} \rangle$.

Combining the above with $\langle H\ ;\ v\ ;\ \#x, S_1 \rangle \overset{\text{Update}}{\longmapsto} \langle H\ ;\ v\ ;\ S_1 \rangle$ then yields the desired result.

4.4.7 Reduction lemma

Another key property of our type system is stated by the reduction lemma. It basically says that if the type of a reduction context and the type of value match up then we can plug the value into the reduction context and perform a reduction. Moreover the possible duplication of free variables is consistent with the types of the variables. This is expressed as follows.

Lemma 4.4.17 (Reduction lemma)
If $\Gamma_0 \vdash R \rightsquigarrow \tilde{R} : [\rho^{[0,0]}]\tau$ and $\Gamma_1 \vdash v \rightsquigarrow \tilde{v} : \rho$ then there exist e and $\tilde{e}$ and Γ such that

- $R[v] \ \mapsto\ e$
- $\tilde{R}[\tilde{v}^\iota] \overset{\sim}{\mapsto} \tilde{e}$
- $\Gamma_0, \Gamma_1 \rightarrow^* \Gamma$
- $\Gamma \vdash e \rightsquigarrow \tilde{e} : \tau$

Proof 4.4.18 (Reduction lemma)
Assume that

$$\Gamma_0 \vdash R \rightsquigarrow \tilde{R} : [\rho^{[0,0]}]\tau$$

and

$$\Gamma_1 \vdash v \rightsquigarrow \tilde{v} : \rho.$$

We proceed by case analysis on R. The cases where $R \equiv [\,] + e$ and $R \equiv \texttt{add}_n\,[\,]$ are trivial (essentially since no variable can be duplicated by the reduction) and have been omitted.

case $R \equiv [\,]\,y$**:** By inspection of the typing rules we see that the derivations of $\Gamma_0 \vdash R \rightsquigarrow \tilde{R} : [\rho^{[0,0]}]\tau$ and $\Gamma_1 \vdash v \rightsquigarrow \tilde{v} : \rho$ are of the following forms.

$$\text{AppR}\frac{}{y : \sigma \vdash [\,]\,y \rightsquigarrow [\,]\,y : [\sigma \to^{[0,0]} \tau]\tau}$$

$$\text{Abs}\frac{\Gamma_1, \Gamma_2 \vdash e \rightsquigarrow \tilde{e} : \tau}{\Gamma_1 \vdash \lambda x.e \rightsquigarrow \lambda x.\tilde{e} : \sigma \to \tau} \quad \begin{array}{l} x \notin \mathrm{dom}(\Gamma_1) \\ x : \sigma \to^* \Gamma_2 \end{array}$$

By definition

$$(\lambda x.e)\; y \;\mapsto\; e[x{:=}y]$$

and

$$(\lambda^{\iota} x.\tilde{e})\; y \;\tilde{\mapsto}\; \tilde{e}[x{:=}y].$$

Now by applying the substitution $[x{:=}y]$ to the side condition $x : \sigma \to^* \Gamma_2$ we know that

$$y : \sigma \to^* \Gamma_2[x{:=}y]$$

so

$$\Gamma_1, y : \sigma \to^* \Gamma_1, \Gamma_2[x{:=}y].$$

It remains to show that $\Gamma_1, \Gamma_2[x{:=}y] \vdash e[x{:=}y] \rightsquigarrow \tilde{e}[x{:=}y] : \tau$. We know that $\Gamma_1, \Gamma_2 \vdash e \rightsquigarrow \tilde{e} : \tau$ and by applying $[x{:=}y]$ to the judgement we get that

$$\Gamma_1[x{:=}y], \Gamma_2[x{:=}y] \vdash e[x{:=}y] \rightsquigarrow \tilde{e}[x{:=}y] : \tau.$$

Now by the side condition $x \notin \mathrm{dom}(\Gamma_1)$ we know that $\Gamma_1[x{:=}y] \equiv \Gamma_1$ so

$$\Gamma_1, \Gamma_2[x{:=}y] \vdash e[x{:=}y] \rightsquigarrow \tilde{e}[x{:=}y] : \tau$$

as required.

case $R \equiv \mathtt{case}\,[\,]\,\mathtt{of}\;alts$**:** By inspection of the typing rules we see that the derivation of $\Gamma_0 \vdash R \rightsquigarrow \tilde{R} : [\rho^{[0,0]}]\tau$ is of the form

$$\text{Case}\frac{\Gamma_2, \Gamma_3, \Gamma_4 \vdash alts \rightsquigarrow \tilde{alts} : \mathtt{List}\,\sigma\,\kappa\,\iota' \Rightarrow \tau}{\Gamma_2, \Gamma_3, \Gamma_4 \vdash \mathtt{case}\,[\,]\,\mathtt{of}\;alts \rightsquigarrow \mathtt{case}\,[\,]\,\mathtt{of}\;\tilde{alts} : [(\mathtt{List}\,\sigma\,\kappa\,\iota')^{[0,0]}]\tau}$$

where $alts = \{\mathtt{nil} \Rightarrow e_0; \mathtt{cons}\,x_0\,x_1 \Rightarrow e_1\}$, $\tilde{alts} = \{\mathtt{nil} \Rightarrow \tilde{e}_0; \mathtt{cons}\,x_0\,x_1 \Rightarrow \tilde{e}_1\}$ and $\Gamma_2, \Gamma_3, \Gamma_4 \vdash alts \rightsquigarrow \tilde{alts} : \mathtt{List}\,\sigma\,\kappa\,\iota' \Rightarrow \tau$ is derived as follows.

$$\text{Alts}\frac{\Gamma_2, \Gamma_3 \vdash e_0 \rightsquigarrow \tilde{e}_0 : \tau \quad \Gamma_2, \Gamma_4, \Gamma_5 \vdash e_1 \rightsquigarrow \tilde{e}_1 : \tau}{\Gamma_2, \Gamma_3, \Gamma_4 \vdash alts \rightsquigarrow \tilde{alts} : \mathtt{List}\,\sigma\,\kappa\,\iota' \Rightarrow \tau} \quad \begin{array}{l} x_0, x_1 \notin \mathrm{dom}(\Gamma_2, \Gamma_4) \\ x_0 : \sigma, x_1 : (\mathtt{List}\,\sigma\,\kappa\,\iota')^{\iota'}_{\kappa} \vdash \Gamma_5 \end{array}$$

Now there are two possible cases. Either $v \equiv \mathtt{nil}$ or $v \equiv \mathtt{cons}\;y_0\;y_1$.

subcase $v \equiv \texttt{nil}$**:** By inspection of the typing rules we see that the derivation of $\Gamma_1 \vdash v \rightsquigarrow \tilde{v} : \rho$ is of the following form.

$$\text{Nil}\frac{}{\vdash \texttt{nil} \rightsquigarrow \texttt{nil} : \texttt{List}\,\sigma\,\kappa\,\iota'}$$

By definition

$$\texttt{case}\,\texttt{nil}\,\texttt{of}\,\{\texttt{nil} \Rightarrow e_0; \texttt{cons}\,x\,y \Rightarrow e_1\} \;\mapsto\; e_0,$$

$$\texttt{case}\,\texttt{nil}^{\iota}\,\texttt{of}\,\{\texttt{nil} \Rightarrow \tilde{e}_0; \texttt{cons}\,x\,y \Rightarrow \tilde{e}_1\} \tilde{\mapsto} \tilde{e}_0$$

and

$$\Gamma_2, \Gamma_3, \Gamma_4 \overset{\text{Drop}}{\rightarrow^*} \Gamma_2, \Gamma_3.$$

We also already have that

$$\Gamma_2, \Gamma_3 \vdash e_0 \rightsquigarrow \tilde{e}_0 : \tau$$

as required.

subcase $v \equiv \texttt{cons}\,y_0\,y_1$**:** By inspection of the typing rules we see that the derivation of $\Gamma_1 \vdash v \rightsquigarrow \tilde{v} : \rho$ is of the following form.

$$\text{Cons}\frac{}{y_0 : \sigma_0, y_1 : \sigma_1 \vdash \texttt{cons}\,y_0\,y_1 \rightsquigarrow \texttt{cons}\,y_0\,y_1 : \texttt{List}\,\sigma\,\kappa\,\iota'} \quad \begin{array}{l}\sigma_0 \leq \sigma \\ \sigma_1 \leq (\texttt{List}\,\sigma\,\kappa\,\iota')^{\iota'}_{\kappa}\end{array}$$

By definition we know that

$$\texttt{case}\,\texttt{cons}\,y_0\,y_1\,\texttt{of}\,\{\texttt{nil} \Rightarrow e_0; \texttt{cons}\,x_0\,x_1 \Rightarrow e_1\} \;\mapsto\; e_1[x_0{:=}y_0, x_1{:=}y_1]$$

and

$$\texttt{case}\,\texttt{cons}^{\iota}\,y_0\,y_1\,\texttt{of}\,\{\texttt{nil} \Rightarrow \tilde{e}_0; \texttt{cons}\,x_0\,x_1 \Rightarrow \tilde{e}_1\} \tilde{\mapsto} \tilde{e}_1[x_0{:=}y_0, x_1{:=}y_1].$$

Now by definition $x_0 : \sigma, x_1 : (\texttt{List}\,\sigma\,\kappa\,\iota')^{\iota'}_{\kappa} \vdash \Gamma_5$ means that

$$x_0 : \sigma, x_1 : (\texttt{List}\,\sigma\,\kappa\,\iota')^{\iota'}_{\kappa} \rightarrow^* \Gamma_5.$$

By applying the substitution $[x_0{:=}y_0, x_1{:=}y_1]$ to the above we know that

$$y_0 : \sigma, y_1 : (\texttt{List}\,\sigma\,\kappa\,\iota')^{\iota'}_{\kappa} \rightarrow^* \Gamma_5[x_0{:=}y_0, x_1{:=}y_1].$$

Now since $\sigma_0 \leq \sigma$ and $\sigma_1 \leq (\texttt{List}\,\sigma\,\kappa\,\iota')^{\iota'}_{\kappa}$ we can use the context rewrite lemma (clause iii) to show that there exist Γ_6 such that

$$\Gamma_6 \leq \Gamma_5[x_0{:=}y_0, x_1{:=}y_1]$$

and

$$y_0 : \sigma_0, y_1 : \sigma_1 \to^* \Gamma_6.$$

Using the latter we get that

$$\Gamma_2, \Gamma_3, \Gamma_4, y_0 : \sigma_0, y_1 : \sigma_1 \to^* \Gamma_2, \Gamma_3, \Gamma_4, \Gamma_6 \overset{\text{Drop}}{\to^*} \Gamma_2, \Gamma_4, \Gamma_6.$$

It remains to show that

$$\Gamma_2, \Gamma_4, \Gamma_6 \vdash e_1[x_0{:=}y_0, x_1{:=}y_1] \rightsquigarrow \tilde{e}_1[x_0{:=}y_0, x_1{:=}y_1] : \tau.$$

We know that $\Gamma_2, \Gamma_4, \Gamma_5 \vdash e_1 \rightsquigarrow \tilde{e}_1 : \tau$ so by applying $[x_0{:=}y_0, x_1{:=}y_1]$ to the judgement we get that

$$(\Gamma_2, \Gamma_4, \Gamma_5)[x_0{:=}y_0, x_1{:=}y_1] \vdash e_1[x_0{:=}y_0, x_1{:=}y_1] \rightsquigarrow \tilde{e}_1[x_0{:=}y_0, x_1{:=}y_1] : \tau.$$

We can use the side condition $x_0, x_1 \notin \mathrm{dom}(\Gamma_2, \Gamma_4)$ to simplify the judgement to

$$\Gamma_2, \Gamma_4, \Gamma_5[x_0{:=}y_0, x_1{:=}y_1] \vdash e_1[x_0{:=}y_0, x_1{:=}y_1] \rightsquigarrow \tilde{e}_1[x_0{:=}y_0, x_1{:=}y_1] : \tau.$$

Finally since $\Gamma_6 \leq \Gamma_5[x_0{:=}y_0, x_1{:=}y_1]$ we may use the subsumption lemma to get

$$\Gamma_2, \Gamma_4, \Gamma_6 \vdash e_1[x_0{:=}y_0, x_1{:=}y_1] \rightsquigarrow \tilde{e}_1[x_0{:=}y_0, x_1{:=}y_1] : \tau$$

as required.

4.4.8 Progress lemma

We now have all the machinery to fill in the postponed proofs. The proofs of source and target transition and the terminal configuration lemma have a lot in common. Thus we will first prove a lemma which implies the properties we need to prove. We call this the progress lemma. It basically says that if C is annotated as $\tilde{C}$ then, modulo updates needed to take care of discarded update markers, either both C and $\tilde{C}$ are values or both are black holes or both can evaluate one step further.

Lemma 4.4.19 (Progress)
If $\vdash C_0 \rightsquigarrow \tilde{C} : \tau$ then there exist $n \geq 0$ and C_i for $1 \leq i \leq n$ such that

- $C_0 \overset{\text{Update}}{\longmapsto} C_1 \overset{\text{Update}}{\longmapsto} \ldots \overset{\text{Update}}{\longmapsto} C_n$
- $\vdash C_i \rightsquigarrow \tilde{C} : \tau$
- Either $C_n \in \text{Value}$ and $\tilde{C} \in \text{Value}^{\sim}$
  or $C_n \in \text{Blackhole}$ and $\tilde{C} \in \text{Blackhole}^{\sim}$
  or $C_n \longmapsto C'$, $\tilde{C} \overset{\sim}{\longmapsto} \tilde{C}'$ and $\vdash C' \rightsquigarrow \tilde{C}' : \tau$.

Proof 4.4.20 (Progress)
Assume that

$$\vdash C \rightsquigarrow \tilde{C} : \tau.$$

We first note that an expression is either a value, a variable, a `let`-expression or of the form $R[e]$. We will use this and make the proof by case analysis on the second component of C. The interesting cases will be when the second component is a value or variable. Simply because these are the cases where the computation can go wrong and fail to proceed.

case $C \equiv \langle H\,;\, v\,;\, S_0\rangle$: Let us start with the case where $C \equiv \langle H\,;\, v\,;\, S_0\rangle$ which turns out to be the most substantial case. Then since the second component of the configuration is a value we know by the discarded update markers lemma that we can take care of any discarded update markers on top of the stack. That is we know that there exist $n \geq 0$ and S_i for $1 \leq i \leq n$ such that:

- $\langle H\,;\, v\,;\, S_0\rangle \overset{\text{Update}}{\longmapsto} \langle H\,;\, v\,;\, S_1\rangle \overset{\text{Update}}{\longmapsto} \ldots \overset{\text{Update}}{\longmapsto} \langle H\,;\, v\,;\, S_n\rangle$
- $\vdash \langle H\,;\, v\,;\, S_i\rangle \rightsquigarrow \tilde{C} : \tau$
- If $S_n \equiv \epsilon$ then $\tilde{C} \equiv \langle \tilde{H}\,;\, \tilde{v}^{\iota}\,;\, \epsilon\rangle$.
- If $S_n \equiv R, S$ then $\tilde{C} \equiv \langle \tilde{H}\,;\, \tilde{v}^{\iota}\,;\, \tilde{R}, \tilde{S}\rangle$.
- If $S_n \equiv \#x, S$ then $\tilde{C} \equiv \langle \tilde{H}\,;\, \tilde{v}^{\iota}\,;\, \#x, \tilde{S}\rangle$.

We proceed by case analysis on the structure of S_n.

subcase $S_n \equiv \epsilon$: We then know that

$$\tilde{C} \equiv \langle \tilde{H}\,;\, \tilde{v}^{\iota}\,;\, \epsilon\rangle$$

so by definition

$$\langle H\,;\, v\,;\, \epsilon\rangle \in \text{Value}$$

and

$$\langle \tilde{H}\,;\, \tilde{v}^{\iota}\,;\, \epsilon\rangle \in \tilde{\text{Value}}.$$

Thus

$$\langle H\,;\, v\,;\, S_0\rangle \overset{\text{Update}}{\longmapsto} \langle H\,;\, v\,;\, S_1\rangle \overset{\text{Update}}{\longmapsto} \ldots \overset{\text{Update}}{\longmapsto} \langle H\,;\, v\,;\, \epsilon\rangle,$$
$$\vdash \langle H\,;\, v\,;\, S_i\rangle \rightsquigarrow \langle \tilde{H}\,;\, \tilde{v}^{\iota}\,;\, \epsilon\rangle : \tau,$$
$$\langle H\,;\, v\,;\, \epsilon\rangle \in \text{Value}$$

and

$$\langle \tilde{H}\,;\, \tilde{v}^{\iota}\,;\, \epsilon\rangle \in \tilde{\text{Value}}$$

as required.

subcase $S_n \equiv R, S$**:** We then know that $\tilde{C} \equiv \langle \tilde{H} \,;\, \tilde{v}^{[\eta,\xi]} \,;\, \tilde{R}, \tilde{S} \rangle$ and by inspection of the typing rules we see that the derivation of $\vdash \langle H \,;\, v \,;\, R, S \rangle \rightsquigarrow \tilde{C} : \tau$ is of the form

$$\text{Config}\frac{\begin{array}{l}\Gamma_0 \vdash H \rightsquigarrow \tilde{H} : \Delta_0 \\ \Gamma_1 \vdash v \rightsquigarrow \tilde{v}^{[\eta,\xi]} : \rho^{[0,0]} \\ \Gamma_2, \Gamma_3 \vdash R, S \rightsquigarrow \tilde{R}, \tilde{S} : \Delta_1 \,;\, [\rho^{[0,0]}]\tau\end{array}}{\vdash \langle H \,;\, v \,;\, R, S \rangle \rightsquigarrow \langle \tilde{H} \,;\, \tilde{v}^{[\eta,\xi]} \,;\, \tilde{R}, \tilde{S} \rangle : \tau} \quad \Delta_0, \Delta_1 \vdash \Gamma_0, \Gamma_1, \Gamma_2, \Gamma_3$$

where

$$\Gamma_1 \vdash v \rightsquigarrow \tilde{v}^{[\eta,\xi]} : \rho^{[0,0]}$$

is derived as

$$\text{Value}\frac{\Gamma_1 \vdash v \rightsquigarrow \tilde{v} : \rho}{\Gamma_1 \vdash v \rightsquigarrow \tilde{v}^{[\eta,\xi]} : \rho^{[0,0]}} \quad \begin{array}{l}\text{if } 0 > 0 \text{ then } \Gamma_1 \to^* \Gamma_1, \Gamma_1 \\ [0,0] \leq [\eta,\xi]\end{array}$$

and

$$\Gamma_2, \Gamma_3 \vdash R, S \rightsquigarrow \tilde{R}, \tilde{S} : \Delta_1 \,;\, [\rho^{[0,0]}]\tau$$

is derived as follows.

$$\text{Stack-R}\frac{\Gamma_2 \vdash R \rightsquigarrow \tilde{R} : [\rho^{[0,0]}]\tau_0 \quad \Gamma_3 \vdash S \rightsquigarrow \tilde{S} : \Delta_1 \,;\, [\tau_0]\tau}{\Gamma_2, \Gamma_3 \vdash R, S \rightsquigarrow \tilde{R}, \tilde{S} : \Delta_1 \,;\, [\rho^{[0,0]}]\tau}$$

Now since $\Gamma_1 \vdash v \rightsquigarrow \tilde{v} : \rho$ and $\Gamma_2 \vdash R \rightsquigarrow \tilde{R} : [\rho^{[0,0]}]\tau_0$ we know by the reduction lemma that there exist e, $\tilde{e}$ and Γ_4 such that

$$R[v] \;\mapsto\; e,$$

$$\tilde{R}[\tilde{v}^{[\eta,\xi]}] \;\tilde{\mapsto}\; \tilde{e},$$

$$\Gamma_1, \Gamma_2 \to^* \Gamma_4$$

and

$$\Gamma_4 \vdash e \rightsquigarrow \tilde{e} : \tau_0.$$

So the only thing that can stop the evaluation of $\langle \tilde{H} \,;\, \tilde{v}^{[\eta,\xi]} \,;\, \tilde{R}, \tilde{S} \rangle$ to proceed is if $\tilde{v}^{[\eta,\xi]}$ requires an update marker on top of the stack. That is if $\eta \neq 0$. But from $[0,0] \leq [\eta,\xi]$ we know immediately that

$$\eta = 0.$$

Thus by definition

$$\langle H \,;\, v \,;\, R, S \rangle \;\longmapsto\; \langle H \,;\, e \,;\, S \rangle$$

and

$$\langle \tilde{H} \,;\, \tilde{v}^{[\eta,\xi]} \,;\, \tilde{R}, \tilde{S} \rangle \;\tilde{\longmapsto}\; \langle \tilde{H} \,;\, \tilde{e} \,;\, \tilde{S} \rangle.$$

It remains to show that $\vdash \langle H\ ;\ e\ ;\ S\rangle \rightsquigarrow \langle \tilde{H}\ ;\ \tilde{e}\ ;\ \tilde{S}\rangle : \tau$. We first show that

$$\Delta_0, \Delta_1 \vdash \Gamma_0, \Gamma_4, \Gamma_3$$

which follows from $\Delta_0, \Delta_1 \vdash \Gamma_0, \Gamma_1, \Gamma_2, \Gamma_3$ and $\Gamma_1, \Gamma_2 \rightarrow^* \Gamma_4$. Now we can derive

$$\vdash \langle H\ ;\ e\ ;\ S\rangle \rightsquigarrow \langle \tilde{H}\ ;\ \tilde{e}\ ;\ \tilde{S}\rangle : \tau$$

as follows.

$$\text{Config}\frac{\begin{array}{l}\Gamma_0 \vdash H \rightsquigarrow \tilde{H} : \Delta_0 \\ \Gamma_4 \vdash e \rightsquigarrow \tilde{e} : \tau_0 \\ \Gamma_3 \vdash S \rightsquigarrow \tilde{S} : \Delta_1\ ;\ [\tau_0]\tau\end{array}}{\vdash \langle H\ ;\ e\ ;\ S\rangle \rightsquigarrow \langle \tilde{H}\ ;\ \tilde{e}\ ;\ \tilde{S}\rangle : \tau} \quad \Delta_0, \Delta_1 \vdash \Gamma_0, \Gamma_4, \Gamma_3$$

To conclude we know that

$$\langle H\ ;\ v\ ;\ S_0\rangle \overset{\text{Update}}{\longmapsto} \langle H\ ;\ v\ ;\ S_1\rangle \overset{\text{Update}}{\longmapsto} \ldots \overset{\text{Update}}{\longmapsto} \langle H\ ;\ v\ ;\ R, S\rangle,$$

$$\vdash \langle H\ ;\ v\ ;\ S_i\rangle \rightsquigarrow \langle \tilde{H}\ ;\ \tilde{v}^{[\eta,\xi]}\ ;\ \tilde{R}, \tilde{S}\rangle : \tau,$$

$$\langle H\ ;\ v\ ;\ R, S\rangle \longmapsto \langle H\ ;\ e\ ;\ S\rangle,$$

$$\langle \tilde{H}\ ;\ \tilde{v}^{[\eta,\xi]}\ ;\ \tilde{R}, \tilde{S}\rangle \tilde{\longmapsto} \langle \tilde{H}\ ;\ \tilde{e}\ ;\ \tilde{S}\rangle$$

and

$$\vdash \langle H\ ;\ e\ ;\ S\rangle \rightsquigarrow \langle \tilde{H}\ ;\ \tilde{e}\ ;\ \tilde{S}\rangle : \tau$$

as required.

subcase $S_n \equiv \#x, S$**:** We then know that $\tilde{C} \equiv \langle \tilde{H}\ ;\ \tilde{v}^{[\eta,\xi]}\ ;\ \#x, \tilde{S}\rangle$ and by inspection of the typing rules we see that the derivation of $\vdash \langle H\ ;\ v\ ;\ \#x, S\rangle \rightsquigarrow \tilde{C} : \tau$ is of the form

$$\text{Config}\frac{\begin{array}{l}\Gamma_0 \vdash H \rightsquigarrow \tilde{H} : \Delta_0 \\ \Gamma_1 \vdash v \rightsquigarrow \tilde{v}^{[\eta,\xi]} : \rho^{[\eta_0+1,\xi_0+1]} \\ \Gamma_2 \vdash \#x, S \rightsquigarrow \#x, \tilde{S} : (\Delta_1, x : \rho_\kappa^{[\eta_0,\xi_0]}\ ;\ [\rho^{[\eta_0+1,\xi_0+1]}]\tau)\end{array}}{\vdash \langle H\ ;\ v\ ;\ \#x, S\rangle \rightsquigarrow \langle \tilde{H}\ ;\ \tilde{v}^{[\eta,\xi]}\ ;\ \#x, \tilde{S}\rangle : \tau} \quad \begin{array}{l}\Delta_0, \Delta_1, x : \rho_\kappa^{[\eta_0,\xi_0]} \\ \vdash \Gamma_0, \Gamma_1, \Gamma_2\end{array}$$

where $\Gamma_1 \vdash v \rightsquigarrow \tilde{v}^{[\eta,\xi]} : \rho^{[\eta_0+1,\xi_0+1]}$ is derived as

$$\text{Value}\frac{\Gamma_1 \vdash v \rightsquigarrow \tilde{v} : \rho}{\Gamma_1 \vdash v \rightsquigarrow \tilde{v}^{[\eta,\xi]} : \rho^{[\eta_0+1,\xi_0+1]}} \quad \begin{array}{l}\text{if } \xi_0 + 1 > 0 \text{ then } \Gamma_1 \rightarrow^* \Gamma_1, \Gamma_1 \\ [\eta_0 + 1, \xi_0 + 1] \leq [\eta, \xi]\end{array}$$

and $\Gamma_2 \vdash \#x, S \rightsquigarrow \#x, \tilde{S} : (\Delta_1, x : \rho_\kappa^{[\eta_0,\xi_0]}\ ;\ [\rho^{[\eta_0+1,\xi_0+1]}]\tau)$ is derived as follows.

$$\text{Stack-}\#\frac{\Gamma_2 \vdash S \rightsquigarrow \tilde{S} : \Delta_1\ ;\ [\tau_0]\tau}{\Gamma_2 \vdash \#x, S \rightsquigarrow \#x, \tilde{S} : (\Delta_1, x : \rho_\kappa^{[\eta_0,\xi_0]}\ ;\ [\rho^{[\eta_0+1,\xi_0+1]}]\tau)} \quad \rho^{[\eta_0,\xi_0]} \leq \tau_0$$

The only thing which can stop $\langle \tilde{H} \ ; \ \tilde{v}^{[\eta,\xi]} \ ; \ \#x, \tilde{S} \rangle$ from evaluating further is if $\tilde{v}^{[\eta,\xi]}$ cannot take care of the update marker. That is if $\xi = 0$. But from $[\eta_0 + 1, \xi_0 + 1] \leq [\eta, \xi]$ we know that

$$\xi > 0.$$

Thus by definition

$$\langle H \ ; \ v \ ; \ \#x, S \rangle \ \longmapsto \ \langle H, x = v \ ; \ v \ ; \ S \rangle$$

and

$$\langle \tilde{H} \ ; \ \tilde{v}^{[\eta,\xi]} \ ; \ \#x, \tilde{S} \rangle \stackrel{\sim}{\longmapsto} \langle \tilde{H}, x =^! \tilde{v}^{[\eta,\xi]} \ ; \ \tilde{v}^{[\eta-1,\xi-1]} \ ; \ \tilde{S} \rangle.$$

It only remains to show that $\vdash \langle H, x{=}v; v; S \rangle \rightsquigarrow \langle \tilde{H}, x{=}^! \tilde{v}^{[\eta,\xi]}; \tilde{v}^{[\eta-1,\xi-1]}; \tilde{S} \rangle : \tau$. We start by deriving

$$\Gamma_1 \vdash x = v \rightsquigarrow x =^! \tilde{v}^{[\eta,\xi]} : (x : \rho_\kappa^{[\eta_0,\xi_0]})$$

as

$$\text{Bind-!} \frac{\Gamma_1 \vdash v \rightsquigarrow \tilde{v}^{[\eta,\xi]} : \rho^{[\eta_0+1,\xi_0+1]}}{\Gamma_1 \vdash x = v \rightsquigarrow x =^! \tilde{v}^{[\eta,\xi]} : (x : \rho_\kappa^{[\eta_0,\xi_0]})}$$

and then derive

$$\Gamma_0, \Gamma_1 \vdash H, x = v \rightsquigarrow \tilde{H}, x =^! \tilde{v}^{[\eta,\xi]} : (\Delta_0, x : \rho_\kappa^{[\eta_0,\xi_0]})$$

as follows.

$$\text{Heap} \frac{\Gamma_0 \vdash H \rightsquigarrow \tilde{H} : \Delta_0 \quad \Gamma_1 \vdash x = v \rightsquigarrow x =^! \tilde{v}^{[\eta,\xi]} : (x : \rho_\kappa^{[\eta_0,\xi_0]})}{\Gamma_0, \Gamma_1 \vdash H, x = v \rightsquigarrow \tilde{H}, x =^! \tilde{v}^{[\eta,\xi]} : (\Delta_0, x : \rho_\kappa^{[\eta_0,\xi_0]})}$$

We know that if $\xi_0 + 1 > 0$ then $\Gamma_1 \rightarrow^* \Gamma_1, \Gamma_1$ so clearly

$$\Gamma_1 \rightarrow^* \Gamma_1, \Gamma_1.$$

and then of course

$$\text{if } \xi_0 > 0 \text{ then } \Gamma_1 \rightarrow^* \Gamma_1, \Gamma_1$$

holds. From $[\eta_0 + 1, \xi_0 + 1] \leq [\eta, \xi]$ we also know that

$$[\eta_0, \xi_0] \leq [\eta - 1, \xi - 1].$$

Now we can derive

$$\Gamma_1 \vdash v \rightsquigarrow \tilde{v}^{[\eta-1,\xi-1]} : \rho^{[\eta_0,\xi_0]}$$

as

$$\text{Value} \frac{\Gamma_1 \vdash v \rightsquigarrow \tilde{v} : \rho}{\Gamma_1 \vdash v \rightsquigarrow \tilde{v}^{[\eta_1-1,\xi_1-1]} : \rho^{[\eta_0,\xi_0]}} \quad \begin{array}{l} \text{if } \xi_0 > 0 \text{ then } \Gamma_1 \rightarrow^* \Gamma_1, \Gamma_1 \\ [\eta_0, \xi_0] \leq [\eta - 1, \xi - 1] \end{array}$$

and by the sumbsumption lemma we then know that

$$\Gamma_1 \vdash v \rightsquigarrow \tilde{v}^{[\eta-1,\xi-1]} : \tau_0.$$

Now from $\Delta_0, \Delta_1, x : \rho_\kappa^{[\eta_0,\xi_0]} \vdash \Gamma_0, \Gamma_1, \Gamma_2$ and $\Gamma_1 \to^* \Gamma_1, \Gamma_1$ we know that

$$\Delta_0, \Delta_1, x : \rho_\kappa^{[\eta_0,\xi_0]} \vdash \Gamma_0, \Gamma_1, \Gamma_1, \Gamma_2$$

Thus we can finally derive

$$\vdash \langle H, x = v \; ; \; v \; ; \; S\rangle \rightsquigarrow \langle \tilde{H}, x =^! \tilde{v}^{[\eta,\xi]} \; ; \; \tilde{v}^{[\eta-1,\xi-1]} \; ; \; \tilde{S}\rangle : \tau$$

as follows.

$$\text{Config}\frac{\begin{array}{l}\Gamma_0, \Gamma_1 \vdash H, x = v \rightsquigarrow \tilde{H}, x =^! \tilde{v}^{[\eta,\xi]} : (\Delta_0, x : \rho_\kappa^{[\eta_0,\xi_0]}) \\ \Gamma_1 \vdash v \rightsquigarrow \tilde{v}^{[\eta-1,\xi-1]} : \tau_0 \\ \Gamma_2 \vdash S \rightsquigarrow \tilde{S} : \Delta_1 \; ; [\tau_0]\tau\end{array}}{\vdash \langle H, x = v \; ; \; v \; ; \; S\rangle \rightsquigarrow \langle \tilde{H}, x =^! \tilde{v}^{[\eta,\xi]} \; ; \; \tilde{v}^{[\eta-1,\xi-1]} \; ; \; \tilde{S}\rangle : \tau} \; \begin{array}{l}\Delta_0, \Delta_1, x : \rho_\kappa^{[\eta_0,\xi_0]} \\ \vdash \Gamma_0, \Gamma_1, \Gamma_1, \Gamma_2\end{array}$$

To conclude we know that

$$\langle H \; ; \; v \; ; \; S_0\rangle \overset{\text{Update}}{\longmapsto} \langle H \; ; \; v \; ; \; S_1\rangle \overset{\text{Update}}{\longmapsto} \ldots \overset{\text{Update}}{\longmapsto} \langle H \; ; \; v \; ; \; \#x, S\rangle$$

$$\vdash \langle H \; ; \; v \; ; \; S_i\rangle \rightsquigarrow \langle \tilde{H} \; ; \; \tilde{v}^{[\eta,\xi]} \; ; \; \#x, \tilde{S}\rangle : \tau$$

$$\langle H \; ; \; v \; ; \; \#x, S\rangle \longmapsto \langle H, x = v \; ; \; v \; ; \; S\rangle$$

$$\langle \tilde{H} \; ; \; \tilde{v}^{[\eta,\xi]} \; ; \; \#x, \tilde{S}\rangle \tilde{\longmapsto} \langle \tilde{H}, x =^! \tilde{v}^{[\eta,\xi]} \; ; \; \tilde{v}^{[\eta-1,\xi-1]} \; ; \; \tilde{S}\rangle$$

and

$$\vdash \langle H, x = v \; ; \; v \; ; \; S\rangle \rightsquigarrow \langle \tilde{H}, x =^! \tilde{v}^{[\eta,\xi]} \; ; \; \tilde{v}^{[\eta-1,\xi-1]} \; ; \; \tilde{S}\rangle : \tau$$

as required. This completes the proof for the case where $C \equiv \langle H \; ; \; v \; ; \; S_0\rangle$.

cases $C \equiv \langle H_0 \; ; \; x \; ; \; S\rangle$, $C \equiv \langle H \; ; \; \texttt{let}\; d\; \texttt{in}\; e \; ; \; S\rangle$ and $C \equiv \langle H \; ; \; R[e] \; ; \; S\rangle$: In the remaining cases, where the second component of the configuration is not a value, the next transition does not at all depend on the stack. Since it does not depend on the stack it does not depend on the discarded update markers either. Thus for these cases we can prove the following.

Either $C \in$ Blackhole and $\tilde{C} \in$ Blackhole$\tilde{}$

or $C \longmapsto C'$, $\tilde{C} \tilde{\longmapsto} \tilde{C}'$ and $\vdash C' \rightsquigarrow \tilde{C}' : \tau$.

This clearly implies the lemma; simply take $n = 0$.

case $C \equiv \langle H_0 \,;\, x \,;\, S\rangle$**:** By inspection of the typing rules we see that the derivation of $\vdash C \rightsquigarrow \tilde{C} : \tau$ is of the form

$$\text{Config}\frac{\begin{array}{c}\Gamma_0 \vdash H_0 \rightsquigarrow \tilde{H}_0 : \Delta_0 \\ x : \tau_{1\,\kappa_0} \vdash x \rightsquigarrow x : \tau_0 \\ \Gamma_1 \vdash S \rightsquigarrow \tilde{S} : \Delta_1 \,;\, [\tau_0]\tau\end{array}}{\vdash \langle H_0 \,;\, x \,;\, S\rangle \rightsquigarrow \langle \tilde{H}_0 \,;\, x \,;\, \tilde{S}\rangle : \tau} \quad \Delta_0, \Delta_1 \vdash \Gamma_0, x : \tau_{1\,\kappa_0}, \Gamma_1$$

where $x : \tau_{1\,\kappa_0} \vdash x \rightsquigarrow x : \tau_0$ is derived as follows.

$$\text{Var}\frac{}{x : \tau_{1\,\kappa_0} \vdash x \rightsquigarrow x : \tau_0} \quad \tau_1 \leq \tau_0$$

Now, since we require configurations to be closed either $x \in \text{dom}(H_0)$ or $x \in \text{dom}(S)$ but not both.

subcase $x \in \text{dom}(S)$**:** By definition

$$\langle H_0 \,;\, x \,;\, S\rangle \in \text{Blackhole}$$

and by the bound variables lemma

$$\text{dom}(\tilde{H}_0) = \text{dom}(H_0)$$

so

$$x \notin \text{dom}(\tilde{H}_0)$$

which implies that

$$\langle \tilde{H}_0 \,;\, x \,;\, \tilde{S}\rangle \in \tilde{\text{Blackhole}}$$

as required.

subcase $x \in \text{dom}(H_0)$**:** Then H_0 then must be of the form $H, x = e$ and the derivation of $\Gamma_0 \vdash H_0 \rightsquigarrow \tilde{H}_0 : \Delta_0$ of the following form.

$$\text{Heap}\frac{\Gamma_2 \vdash H \rightsquigarrow \tilde{H} : \Delta_2 \quad \Gamma_3 \vdash x = e \rightsquigarrow x =^{\kappa} \tilde{e} : (x : \sigma)}{\Gamma_2, \Gamma_3 \vdash H, x = e \rightsquigarrow \tilde{H}, x =^{\kappa} \tilde{e} : (\Delta_2, x : \sigma)}$$

By now we know by the forms of the derivations that $C \equiv \langle H, x = e \,;\, x \,;\, S\rangle$ and $\tilde{C} \equiv \langle \tilde{H}, x =^{\kappa} \tilde{e} \,;\, x \,;\, \tilde{S}\rangle$. We proceed by cases on κ.

subsubcase $\kappa = \checkmark$**:** Then the derivation of $\Gamma_3 \vdash x = e \rightsquigarrow x =^{\kappa} \tilde{e} : (x : \sigma)$ must be of the following form.

$$\text{Bind-}\checkmark\frac{\Gamma_3 \vdash e \rightsquigarrow \tilde{e} : \tau_2}{\Gamma_3 \vdash x = e \rightsquigarrow x =^{\checkmark} \tilde{e} : (x : \tau_{2\checkmark})}$$

Now by definition

$$\langle H, x = e\,;\; x\,;\; S\rangle \;\longmapsto\; \langle H\,;\; e\,;\; \#x, S\rangle$$

and we need to show that $\langle \tilde{H}, x =^{\checkmark} \tilde{e}\,;\; x\,;\; \tilde{S}\rangle \tilde{\longmapsto} \langle \tilde{H}\,;\; \tilde{e}\,;\; \tilde{S}\rangle$, ie that it is safe to not push an update marker for x. It is safe if we can show that we do not create any dangling pointers. That is we need to show that $\langle \tilde{H}\,;\; \tilde{e}\,;\; \tilde{S}\rangle$ remains closed which amounts to showing that $x \notin \mathrm{fv}(\tilde{H}) \cup \mathrm{fv}(\tilde{e}) \cup \mathrm{fv}(\tilde{S})$. From the derivation of $\Gamma_0 \vdash H_0 \rightsquigarrow \tilde{H}_0 : \Delta_0$ we know that

$$\Delta_0 \equiv \Delta_2, x : \sigma$$

and

$$\Gamma_0 \equiv \Gamma_2, \Gamma_3$$

and from the derivation of $\Gamma_1 \vdash x = e \rightsquigarrow x =^{\kappa} \tilde{e} : x : \sigma$ we know that

$$\sigma \equiv \tau_{2\checkmark}.$$

Using $\Delta_0, \Delta_1 \vdash \Gamma_0, x : \tau_{1\kappa_0}, \Gamma_1$ and these facts we know that

$$\Delta_2, x : \tau_{2\checkmark}, \Delta_1 \vdash \Gamma_2, \Gamma_3, x : \tau_{1\kappa_0}, \Gamma_1.$$

By the context entailment lemma (clause iv) we then get that

$$\tau_{2\checkmark} \equiv \tau_{1\kappa_0}$$

and thus

$$\Delta_2, x : \tau_{2\checkmark}, \Delta_1 \vdash \Gamma_2, \Gamma_3, x : \tau_{2\checkmark}, \Gamma_1$$

so

$$\Delta_2, \Delta_1 \vdash \Gamma_2, \Gamma_3, \Gamma_1$$

by the context entailment lemma (clause v). Now since $\Delta_2, x : \tau_{2\checkmark}, \Delta_1$ is a distinct context we know that

$$x \notin \mathrm{dom}(\Delta_2, \Delta_1)$$

and thus by $\Delta_2, \Delta_1 \vdash \Gamma_2, \Gamma_3, \Gamma_1$

$$x \notin \mathrm{dom}(\Gamma_2, \Gamma_3, \Gamma_1).$$

Now, since we can type $\tilde{H}$, $\tilde{e}$ and $\tilde{S}$ in Γ_2, Γ_3 and Γ_1 respectively, we know by the free variables lemma that

$$x \notin \mathrm{fv}(\tilde{H}) \cup \mathrm{fv}(\tilde{e}) \cup \mathrm{fv}(\tilde{S})$$

Thus we may conclude that $\langle \tilde{H}\,;\; \tilde{e}\,;\; \tilde{S}\rangle$ is closed and that

$$\langle \tilde{H}, x =^{\checkmark} \tilde{e}\,;\; x\,;\; \tilde{S}\rangle \tilde{\longmapsto} \langle \tilde{H}\,;\; \tilde{e}\,;\; \tilde{S}\rangle.$$

It only remains to show that $\vdash \langle H\ ;\ e\ ;\ \#x, S\rangle \rightsquigarrow \langle \tilde{H}\ ;\ \tilde{e}\ ;\ \tilde{S}\rangle : \tau$. We start by showing that

$$\Gamma_1 \vdash \#x, S \rightsquigarrow \tilde{S} : \Delta_1\ ;\ [\tau_2]\tau$$

which can be derived as

$$\text{Stack-\#-discard}\frac{\Gamma_1 \vdash S \rightsquigarrow \tilde{S} : \Delta_1\ ;\ [\tau_0]\tau}{\Gamma_1 \vdash \#x, S \rightsquigarrow \tilde{S} : \Delta_1\ ;\ [\tau_2]\tau} \quad \tau_2 \leq \tau_0$$

where we obtain $\tau_2 \leq \tau_0$ from $\tau_1 \leq \tau_0$ and $\tau_2 \equiv \tau_1$. We can now finally derive

$$\vdash \langle H\ ;\ e\ ;\ \#x, S\rangle \rightsquigarrow \langle \tilde{H}\ ;\ \tilde{e}\ ;\ \tilde{S}\rangle : \tau$$

as follows.

$$\text{Config}\frac{\begin{array}{l}\Gamma_2 \vdash H \rightsquigarrow \tilde{H} : \Delta_2 \\ \Gamma_3 \vdash e \rightsquigarrow \tilde{e} : \tau_2 \\ \Gamma_1 \vdash \#x, S \rightsquigarrow \tilde{S} : \Delta_1\ ;\ [\tau_2]\tau\end{array}}{\vdash \langle H\ ;\ e\ ;\ \#x, S\rangle \rightsquigarrow \langle \tilde{H}\ ;\ \tilde{e}\ ;\ \tilde{S}\rangle : \tau} \quad \Delta_2, \Delta_1 \vdash \Gamma_2, \Gamma_3, \Gamma_1$$

subsubcase $\kappa = !$: In this case $\tilde{C} \equiv \langle \tilde{H}, x =^{!} \tilde{e}\ ;\ x\ ;\ \tilde{S}\rangle$ and the derivation of $\Gamma_3 \vdash x = e \rightsquigarrow x =^{\kappa} \tilde{e} : (x : \sigma)$ must be of the following form.

$$\text{Bind-!}\frac{\Gamma_3 \vdash e \rightsquigarrow \tilde{e} : \rho^{[\eta+1, \xi+1]}}{\Gamma_3 \vdash x = e \rightsquigarrow x =^{!} \tilde{e} : x : \rho_{\kappa_1}^{[\eta,\xi]}}$$

By definition

$$\langle H, x = e\ ;\ x\ ;\ S\rangle \longmapsto \langle H\ ;\ e\ ;\ \#x, S\rangle$$

and

$$\langle \tilde{H}, x =^{!} \tilde{e}\ ;\ x\ ;\ \tilde{S}\rangle \tilde{\longmapsto} \langle \tilde{H}\ ;\ \tilde{e}\ ;\ \#x, \tilde{S}\rangle$$

so we only need to show that $\vdash \langle H\ ;\ e\ ;\ \#x, S\rangle \rightsquigarrow \langle \tilde{H}\ ;\ \tilde{e}\ ;\ \#x, \tilde{S}\rangle : \tau$. From the derivation of $\Gamma_0 \vdash H_0 \rightsquigarrow \tilde{H}_0 : \Delta_0$ we know that

$$\Delta_0 \equiv \Delta_2, x : \sigma$$

and

$$\Gamma_0 \equiv \Gamma_2, \Gamma_3$$

and from the derivation of $\Gamma_1 \vdash x = e \rightsquigarrow x =^{\kappa} \tilde{e} : x : \sigma$ we know that

$$\sigma \equiv \rho_{\kappa_1}^{[\eta,\xi]}.$$

Using $\Delta_0, \Delta_1 \vdash \Gamma_0, x : \tau_{1_{\kappa_0}}, \Gamma_1$ and these facts we know that

$$\Delta_2, x : \rho_{\kappa_1}^{[\eta,\xi]}, \Delta_1 \vdash \Gamma_2, \Gamma_3, x : \tau_{1_{\kappa_0}}, \Gamma_1.$$

From this follows by the context entailment lemma (clause iv) that

$$\rho^{[\eta,\xi]}_{\kappa_1} \equiv \tau_{1\kappa_0}$$

and thus from $\tau_1 \leq \tau_0$ we know that

$$\rho^{[\eta,\xi]} \leq \tau_0.$$

From $\Delta_2, x : \rho^{[\eta,\xi]}_{\kappa_1}, \Delta_1 \vdash \Gamma_2, \Gamma_3, x : \tau_{1\kappa_0}, \Gamma_1$ it also follows that

$$\Delta_2, x : \rho^{[\eta,\xi]}_{\kappa_1}, \Delta_1 \vdash \Gamma_2, \Gamma_3, \Gamma_1$$

since $\Gamma_2, \Gamma_3, x : \tau_{1\kappa_0}, \Gamma_1 \overset{\text{Drop}}{\rightarrow} \Gamma_2, \Gamma_3, \Gamma_1$. We can now build a derivation of

$$\Gamma_1 \vdash \#x, S \rightsquigarrow \#x, \tilde{S} : \Delta_1, x : \rho^{[\eta,\xi]}_{\kappa_1} ; [\rho^{[\eta+1,\xi+1]}]\tau$$

as follows.

$$\text{Stack-}\#\frac{\Gamma_1 \vdash S \rightsquigarrow \tilde{S} : \Delta_1 ; [\tau_0]\tau}{\Gamma_1 \vdash \#x, S \rightsquigarrow \#x, \tilde{S} : \Delta_1, x : \rho^{[\eta,\xi]}_{\kappa_1} ; [\rho^{[\eta+1,\xi+1]}]\tau} \quad \rho^{[\eta,\xi]} \leq \tau_0$$

And finally $\vdash \langle H \,;\, e \,;\, \#x, S\rangle \rightsquigarrow \langle \tilde{H} \,;\, \tilde{e} \,;\, \#x, \tilde{S}\rangle : \tau$ can be derived as follows.

$$\text{Config}\frac{\begin{array}{l}\Gamma_2 \vdash H \rightsquigarrow \tilde{H} : \Delta_2 \\ \Gamma_3 \vdash e \rightsquigarrow \tilde{e} : \rho^{[\eta+1,\xi+1]} \\ \Gamma_1 \vdash \#x, S \rightsquigarrow \#x, \tilde{S} : \Delta_1, x : \rho^{[\eta,\xi]}_{\kappa_1} ; [\rho^{[\eta+1,\xi+1]}]\tau\end{array}}{\vdash \langle H \,;\, e \,;\, \#x, S\rangle \rightsquigarrow \langle \tilde{H} \,;\, \tilde{e} \,;\, \#x, \tilde{S}\rangle : \tau} \quad \Delta_2, \Delta_1, x : \rho^{[\eta,\xi]}_{\kappa_1} \vdash \Gamma_2, \Gamma_3, \Gamma_1$$

This concludes the case where $C \equiv \langle H_0 \,;\, x \,;\, S\rangle$.

case $C \equiv \langle H \,;\, \texttt{let}\, d\, \texttt{in}\, e \,;\, S\rangle$**:** By inspection of the typing rules we see that the derivation of $\vdash C \rightsquigarrow \tilde{C} : \tau$ is of the form

$$\text{Config}\frac{\begin{array}{l}\Gamma_0 \vdash H \rightsquigarrow \tilde{H} : \Delta_0 \\ \Gamma_1, \Gamma_3 \vdash \texttt{let}\, d\, \texttt{in}\, e \rightsquigarrow \texttt{let}\, \tilde{d}\, \texttt{in}\, \tilde{e} : \tau_0 \\ \Gamma_5 \vdash S \rightsquigarrow \tilde{S} : \Delta_2 ; [\tau_0]\tau\end{array}}{\vdash \langle H \,;\, \texttt{let}\, d\, \texttt{in}\, e \,;\, S\rangle \rightsquigarrow \langle \tilde{H} \,;\, \texttt{let}\, \tilde{d}\, \texttt{in}\, \tilde{e} \,;\, \tilde{S}\rangle : \tau} \quad \Delta_0, \Delta_2 \vdash \Gamma_0, \Gamma_1, \Gamma_3, \Gamma_5$$

where $\Gamma_1, \Gamma_3 \vdash \texttt{let}\, d\, \texttt{in}\, e \rightsquigarrow \texttt{let}\, \tilde{d}\, \texttt{in}\, \tilde{e} : \tau_0$ is derived as follows.

$$\text{Let}\frac{\Gamma_1, \Gamma_2 \vdash d \rightsquigarrow \tilde{d} : \Delta_1 \quad \Gamma_3, \Gamma_4 \vdash e \rightsquigarrow \tilde{e} : \tau_0}{\Gamma_1, \Gamma_3 \vdash \texttt{let}\, d\, \texttt{in}\, e \rightsquigarrow \texttt{let}\, \tilde{d}\, \texttt{in}\, \tilde{e} : \tau_0} \quad \begin{array}{l}\text{dom}(\tilde{d}) \cap \text{dom}(\Gamma_1, \Gamma_3) = \emptyset \\ \Delta_1 \vdash \Gamma_2, \Gamma_4\end{array}$$

Without loss of generality we assume that

$$\text{dom}(d) \cap (\text{dom}(H) \cup \text{dom}(S)) = \emptyset$$

This implies by the bound variables lemma that also

$$\mathrm{dom}(\tilde{d}) \cap (\mathrm{dom}(\tilde{H}) \cup \mathrm{dom}(\tilde{S})) = \emptyset$$

and

$$\mathrm{dom}(\Delta_1) \cap (\mathrm{dom}(\Delta_0) \cup \mathrm{dom}(\Delta_2)) = \emptyset.$$

Thus by definition

$$\langle H \,;\, \mathtt{let}\, d\, \mathtt{in}\, e \,;\, S\rangle \longmapsto \langle H, d \,;\, e \,;\, S\rangle$$

and

$$\langle \tilde{H} \,;\, \mathtt{let}\, \tilde{d}\, \mathtt{in}\, \tilde{e} \,;\, \tilde{S}\rangle \tilde{\longmapsto} \langle \tilde{H}, \tilde{d} \,;\, \tilde{e} \,;\, \tilde{S}\rangle$$

so it only remains to show that $\vdash \langle H, d \,;\, e \,;\, S\rangle \rightsquigarrow \langle \tilde{H}, \tilde{d} \,;\, \tilde{e} \,;\, \tilde{S}\rangle : \tau$. We start by showing that

$$\Gamma_0, \Gamma_1, \Gamma_2 \vdash H, d \rightsquigarrow \tilde{H}, \tilde{d} : \Delta_0, \Delta_1$$

which follows from $\Gamma_0 \vdash H \rightsquigarrow \tilde{H} : \Delta_0$ and $\Gamma_1, \Gamma_2 \vdash d \rightsquigarrow \tilde{d} : \Delta_1$ by a simple induction over the size of d. The next step is to show that

$$\Delta_0, \Delta_1, \Delta_2 \vdash \Gamma_0, \Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4, \Gamma_5$$

which follows from $\mathrm{dom}(\Delta_1) \cap (\mathrm{dom}(\Delta_0) \cup \mathrm{dom}(\Delta_2)) = \emptyset$, $\Delta_0, \Delta_2 \vdash \Gamma_0, \Gamma_1, \Gamma_3, \Gamma_5$ and $\Delta_1 \vdash \Gamma_2, \Gamma_4$ by the context entailment lemma (clause iii). Now we can derive

$$\vdash \langle H, d \,;\, e \,;\, S\rangle \rightsquigarrow \langle \tilde{H}, \tilde{d} \,;\, \tilde{e} \,;\, \tilde{S}\rangle : \tau$$

as follows.

$$\mathrm{Config}\frac{\begin{array}{l}\Gamma_0, \Gamma_1, \Gamma_2 \vdash H, d \rightsquigarrow \tilde{H}, \tilde{d} : \Delta_0, \Delta_1 \\ \Gamma_3, \Gamma_4 \vdash e \rightsquigarrow \tilde{e} : \tau_0 \\ \Gamma_5 \vdash S \rightsquigarrow \tilde{S} : \Delta_2 \,;\, [\tau_0]\tau\end{array}}{\vdash \langle H, d \,;\, e \,;\, S\rangle \rightsquigarrow \langle \tilde{H}, \tilde{d} \,;\, \tilde{e} \,;\, \tilde{S}\rangle : \tau} \quad \Delta_0, \Delta_1, \Delta_2 \vdash \Gamma_0, \Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4, \Gamma_5$$

This concludes the case where $C \equiv \langle H \,;\, \mathtt{let}\, d\, \mathtt{in}\, e \,;\, S\rangle$.

case $C \equiv \langle H \,;\, R[e] \,;\, S\rangle$**:** By inspection of the typing rules we see that the derivation of $\vdash C \rightsquigarrow \tilde{C} : \tau$ is of the form

$$\mathrm{Config}\frac{\begin{array}{l}\Gamma_0 \vdash H \rightsquigarrow \tilde{H} : \Delta_0 \\ \Gamma_1 \vdash R[e] \rightsquigarrow \tilde{R}[\tilde{e}] : \tau_0 \\ \Gamma_2 \vdash S \rightsquigarrow \tilde{S} : \Delta_1 \,;\, [\tau_0]\tau\end{array}}{\vdash \langle H \,;\, R[e] \,;\, S\rangle \rightsquigarrow \langle \tilde{H} \,;\, \tilde{R}[\tilde{e}] \,;\, \tilde{S}\rangle : \tau} \quad \Delta_0, \Delta_1 \vdash \Gamma_0, \Gamma_1, \Gamma_2$$

By definition

$$\langle H \,;\, R[e] \,;\, S\rangle \longmapsto \langle H \,;\, e \,;\, R, S\rangle$$

and

$$\langle \tilde{H} \,;\, \tilde{R}[\tilde{e}] \,;\, \tilde{S} \rangle \tilde{\longmapsto} \langle \tilde{H} \,;\, \tilde{e} \,;\, \tilde{R}, \tilde{S} \rangle$$

so it only remains to show that $\vdash \langle H \,;\, e \,;\, R, S \rangle \rightsquigarrow \langle \tilde{H} \,;\, \tilde{e} \,;\, \tilde{R}, \tilde{S} \rangle : \tau$. Now by the unwind lemma and $\Gamma_1 \vdash R[e] \rightsquigarrow \tilde{R}[\tilde{e}] : \tau_0$ we get that

$$\Gamma_3 \vdash R \rightsquigarrow \tilde{R} : [\rho^{[0,0]}]\tau_0$$

and

$$\Gamma_4 \vdash e \rightsquigarrow \tilde{e} : \rho^{[0,0]}$$

where

$$\Gamma_1 \equiv \Gamma_3, \Gamma_4.$$

We can now derive

$$\Gamma_3, \Gamma_2 \vdash R, S \rightsquigarrow \tilde{R}, \tilde{S} : \Delta_1 \,;\, [\rho^{[0,0]}]\tau$$

as follows.

$$\text{Stack-R}\frac{\Gamma_3 \vdash R \rightsquigarrow \tilde{R} : [\rho^{[0,0]}]\tau_0 \quad \Gamma_2 \vdash S \rightsquigarrow \tilde{S} : \Delta_1 \,;\, [\tau_0]\tau}{\Gamma_3, \Gamma_2 \vdash R, S \rightsquigarrow \tilde{R}, \tilde{S} : \Delta_1 \,;\, [\rho^{[0,0]}]\tau}$$

From $\Delta_0, \Delta_1 \vdash \Gamma_0, \Gamma_1, \Gamma_2$ and $\Gamma_1 \equiv \Gamma_3, \Gamma_4$ we know that

$$\Delta_0, \Delta_1 \vdash \Gamma_0, \Gamma_4, \Gamma_3, \Gamma_2.$$

so we can derive

$$\vdash \langle H \,;\, e \,;\, R, S \rangle \rightsquigarrow \langle \tilde{H} \,;\, \tilde{e} \,;\, \tilde{R}, \tilde{S} \rangle : \tau$$

as follows.

$$\text{Config}\frac{\begin{array}{l}\Gamma_0 \vdash H \rightsquigarrow \tilde{H} : \Delta_0 \\ \Gamma_4 \vdash e \rightsquigarrow \tilde{e} : \tau_1 \\ \Gamma_3, \Gamma_2 \vdash R, S \rightsquigarrow \tilde{R}, \tilde{S} : \Delta_1 \,;\, [\tau_1]\tau\end{array}}{\vdash \langle H \,;\, e \,;\, R, S \rangle \rightsquigarrow \langle \tilde{H} \,;\, \tilde{e} \,;\, \tilde{R}, \tilde{S} \rangle : \tau} \quad \Delta_0, \Delta_1 \vdash \Gamma_0, \Gamma_4, \Gamma_3, \Gamma_2$$

This concludes the proof of the progress lemma.

4.4.9 Proof of source transition

We are now ready to make the postponed proofs. We will start by the source transition property.

Proposition 4.4.21 (Source transition)
If $\vdash C \rightsquigarrow \tilde{C} : \tau$ and $C \longmapsto C'$ then there exists $\tilde{C}'$ such that

- $\tilde{C} \tilde{\longmapsto}^{0/1} \tilde{C}'$

- $\vdash C' \rightsquigarrow \tilde{C}' : \tau$.

Proof 4.4.22 (Source transition)
Assume $\vdash C \rightsquigarrow \tilde{C} : \tau$ and $C \longmapsto C'$. Then by the progress lemma we know that there exist $n \geq 0$ and C_i for $1 \leq i \leq n$ such that

- $C \overset{\text{Update}}{\longmapsto} C_1 \overset{\text{Update}}{\longmapsto} \ldots \overset{\text{Update}}{\longmapsto} C_n$
- $\vdash C_i \rightsquigarrow \tilde{C} : \tau$
- Either $C_n \in \text{Value}$ and $\tilde{C} \in \text{Value}^{\sim}$.
  or $C_n \in \text{Blackhole}$ and $\tilde{C} \in \text{Blackhole}^{\sim}$
  or $C_n \longmapsto C'$, $\tilde{C} \overset{\sim}{\longmapsto} \tilde{C}'$ and $\vdash C' \rightsquigarrow \tilde{C}' : \tau$

Now if $n > 0$ we know that $C' \equiv C_1$ so $\vdash C' \rightsquigarrow \tilde{C} : \tau$ and of course $\tilde{C} \overset{\sim}{\longmapsto}^{0/1} \tilde{C}$ as required. Suppose instead that $n = 0$. Then since neither $C \in \text{Value}$ nor $C \in \text{Blackhole}$ we know there exists $\tilde{C}'$ such that $\tilde{C} \overset{\sim}{\longmapsto} \tilde{C}'$ and $\vdash C' \rightsquigarrow \tilde{C}' : \tau$ as required.

4.4.10 Proof of target transition

In very much the same way we prove the target transition property.

Proposition 4.4.23 (Target transition)
If $\vdash C \rightsquigarrow \tilde{C} : \tau$ and $\tilde{C} \overset{\sim}{\longmapsto} \tilde{C}'$ then there exists C' such that

- $C \longmapsto^{+} C'$
- $\vdash C' \rightsquigarrow \tilde{C}' : \tau$.

Proof 4.4.24 (Target transition)
Assume $\vdash C \rightsquigarrow \tilde{C} : \tau$ and $\tilde{C} \overset{\sim}{\longmapsto} \tilde{C}'$. By the progress lemma we know there exist $n \geq 0$ and C_i for $1 \leq i \leq n$ such that

- $C \overset{\text{Update}}{\longmapsto} C_1 \overset{\text{Update}}{\longmapsto} \ldots \overset{\text{Update}}{\longmapsto} C_n$
- $\vdash C_i \rightsquigarrow \tilde{C} : \tau$
- Either $C_n \in \text{Value}$ and $\tilde{C} \in \text{Value}^{\sim}$.
  or $C_n \in \text{Blackhole}$ and $\tilde{C} \in \text{Blackhole}^{\sim}$
  or $C_n \longmapsto C'$, $\tilde{C} \overset{\sim}{\longmapsto} \tilde{C}'$ and $\vdash C' \rightsquigarrow \tilde{C}' : \tau$

Since neither $\tilde{C} \in \text{Value}^{\sim}$ nor $\tilde{C} \in \text{Blackhole}^{\sim}$ we know that $C_n \longmapsto C'$ and $\vdash C' \rightsquigarrow \tilde{C}' : \tau$. Thus $C \longmapsto^{+} C'$ as required.

4.4.11 Proof of the terminal configuration property

Finally we prove the terminal configuration property.

Lemma 4.4.25 (Terminal configuration property)
If $\vdash C \rightsquigarrow \tilde{C} : \tau$ then

(i) If $C \in \text{Value}$ then $\tilde{C} \in \text{Value}^{\sim}$

(ii) If $\tilde{C} \in \text{Value}^{\sim}$ then $C \longmapsto^* V$ and $\vdash V \rightsquigarrow \tilde{C} : \tau$

(iii) $C \in \text{Blackhole}$ iff $\tilde{C} \in \text{Blackhole}^{\sim}$

(iv) $C \notin \text{Wrong}$ and $\tilde{C} \notin \text{Wrong}^{\sim}$

Proof 4.4.26 (Terminal configuration lemma)
All the clauses are proved easily in a similar manner so we will only prove clause (ii). Assume that $\vdash C \rightsquigarrow \tilde{C} : \tau$ and $\tilde{C} \in \text{Value}^{\sim}$. Then by the progress lemma there exist $n \geq 0$ and C_i for $1 \leq i \leq n$ such that

- $C \overset{\text{Update}}{\longmapsto} C_1 \overset{\text{Update}}{\longmapsto} \ldots \overset{\text{Update}}{\longmapsto} C_n$
- $\vdash C_i \rightsquigarrow \tilde{C} : \tau$
- Either $C_n \in \text{Value}$ and $\tilde{C} \in \text{Value}^{\sim}$.
  or $C_n \in \text{Blackhole}$ and $\tilde{C} \in \text{Blackhole}^{\sim}$
  or $C_n \longmapsto C'$, $\tilde{C} \overset{\sim}{\longmapsto} \tilde{C}'$ and $\vdash C' \rightsquigarrow \tilde{C}' : \tau$

Since we know that $\tilde{C} \in \text{Value}^{\sim}$ it must be the case that $C_n \in \text{Value}$ and we already know that $C \longmapsto^* C_n$ and $\vdash C_n \rightsquigarrow \tilde{C} : \tau$ as required.

Chapter 5

Implementation

In this chapter we will describe and argue the correctness of our implementation of the type system.

5.1 The best annotations

Given an unannotated term it is the task of the implementation to find a corresponding well-typed annotated term. However, usually several well-typed annotated terms can be obtained from a single unannotated term and the implementation should naturally choose the best one. That is the one which avoids as many updates and update marker checks as possible. It is however not always clear which annotated term to choose, as illustrated by the following example.

$$\begin{array}{l}\lambda^{[0,0]}x.\texttt{let}\ y =^{\checkmark} 1^{[0,0]} +^{[0,1]} 2^{[0,0]}\ \texttt{in}\\ \quad \texttt{case}\ x\ \texttt{of}\\ \quad\quad \texttt{nil} \Rightarrow \texttt{let}\ p =^{!} id\ y\ \texttt{in}\ p +^{[0,0]} p\\ \quad\quad \texttt{cons}\ z\ zs \Rightarrow \texttt{let}\ q =^{\checkmark} id\ y\ \texttt{in}\ z +^{[0,0]} q\end{array}$$

Here, $1 + 2$ is annotated with $[0,1]$ because there will be one update marker on top of the stack if the `nil`-branch is taken and none if the `cons`-branch is taken. This means that an update marker check has to be performed when $1+2$ has been computed. However, if we annotate the binding of q with ! there will always be exactly one update marker and we can annotate $1 + 2$ with $[1,1]$ instead. Thus, we can avoid the marker check at the expense of performing an extra update when the `cons`-branch is taken. To choose the best of these alternatives the relative cost of updates and update marker checks as well as the relative frequency of the `case`-branches have to be taken into consideration. We will defer such choices and first minimise the number of updates and then minimise the number of marker checks. We will define an ordering on terms which reflects this intention. First, let $\vec{\kappa}$ and $\vec{\iota}$ range over vectors of κ's and ι's

respectively. For vectors of length n, let $\vec{\kappa} \leq \vec{\kappa}'$ iff $\vec{\kappa}_i \leq \vec{\kappa}'_i$ for all $1 \leq i \leq n$. Let $\vec{\iota} \leq \vec{\iota}'$ be defined analogously and let $(\vec{\kappa}, \vec{\iota}) \preceq (\vec{\kappa}', \vec{\iota}')$ be the lexicographic ordering on pairs induced by these two orderings. For every annotated term $\tilde{e}$ we associate a pair $(\vec{\kappa}, \vec{\iota})$ consisting of the annotations on the term (in some order given by the structure of $\tilde{e}$). Let $\tilde{e}$ and $\tilde{e}'$ be two terms with the same underlying structure (ie if we remove all annotations the terms become equal). Let $(\vec{\kappa}', \vec{\iota}')$ and $(\vec{\kappa}', \vec{\iota}')$ be the annotations associated with $\tilde{e}$ and $\tilde{e}'$ respectively. We will then write $\tilde{e} \preceq \tilde{e}'$ iff $(\vec{\kappa}, \vec{\iota}) \preceq (\vec{\kappa}', \vec{\iota}')$ and we will say that $\tilde{e}$ is better annotated than $\tilde{e}'$. Thus the objective of our algorithm will be to find the smallest well-typed term according to this ordering.

5.2 Overview of the implementation

We will divide our implementation in three distinct phases. The first phase of the implementation will perform an ordinary type inference based on an ordinary monomorphic type system. This approach has several advantages. First ordinary type inference is well understood and there are efficient implementations. Second, most compilers performs ordinary type inference anyway and sometimes keep type information to later passes by means of an explicitly typed intermediate language.

The second phase of the implementation will use the information produced by the first phase and produce an typing judgement in our type system as its output. Unfortunately the type system of chapter 3 does not enjoy a principal typing property, ie there is no typing judgement that represents all typing judgements. Thus we will modify the type system of chapter 3 to achieve a principal typing property. This involves extending the term and the type language with annotation variables and type variables and it also necessitates the introduction of constraints constraining the type variables and annotation variables.

Finally the third phase of the implementation will take the constraint set generated by the second phase as its input and compute a solution. This solution then generates the best well-typed annotated term.

It should be pointed out that the algorithm is global in the sense that it requires access to the entire program. Thus the implementation is not well suited for separate compilation. This is inherent in the type system since the annotations on a function can very well influence the annotations on functions in other modules. In chapter 8 we will discuss some ideas of how to attack the problem of separate compilation.

5.3 The underlying type system

In this section we will present the underlying ordinary type system that the first phase of our algorithm works with. We will use an explicitly typed term language which allows us to easily propagate type information to the second

phase of our algorithm. However, we first define our ordinary type language as follows.

Type variables	a, b, c	
Types	δ ::=	$a \mid \texttt{Int} \mid \delta_0 \to \delta_1 \mid \texttt{List}\ \delta$
Contexts	Ω ::=	$x_1 : \delta_1,\ ...,\ x_n : \delta_n$
		where $x_i \neq x_j$ if $i \neq j$

Now we can define the term language as follows.

Variables	x, y, z	
Values	$\underline{v}$ ::=	$\lambda x.\underline{e} \mid n \mid \texttt{nil} \mid \texttt{cons}\ x\ y$
Expressions	$\underline{e}$ ::=	$\underline{v} \mid x \mid \underline{e}\ x \mid \underline{e}_0 + \underline{e}_1 \mid$
		$\texttt{let}\ \underline{d} : \Omega\ \texttt{in}\ \underline{e} \mid$
		$\texttt{case}\ \underline{e} : \delta\ \texttt{of}\ \underline{alts}$
Declarations	$\underline{d}$ ::=	$\epsilon \mid \underline{d},\ \underline{b}$
Bindings	$\underline{b}$ ::=	$x = \underline{e}$
Alternatives	$\underline{alts}$ ::=	$\{\texttt{nil} \Rightarrow \underline{e}_0; \texttt{cons}\ x\ y \Rightarrow \underline{e}_1\}$

We have added type annotations to the language such that it is easy to (given arbitrary Ω, $\underline{e}$ and δ) check whether $\Omega \vdash \underline{e} : \tau$ (which will be defined shortly) holds or not. Since the types of the bindings in a `let`-expressions are not necessarily visible in the type of the `let`-expression we need to annotate the `let`-expression with a context Ω giving the types of the bindings. Similarly, we need to annotate each `case`-expression with the type of the term it scrutinises. Note that we do not need to annotate λ-abstractions due to the restricted form of application in our language. By not doing so we avoid redundancy which would complicate our proofs considerably. Note that we will let $\underline{e}$ range over explicitly typed terms to avoid confusion with the untyped terms introduced in chapter 2 which are ranged over by e. We will write $\lfloor \underline{e} \rfloor$ for the untyped term obtained by removing type annotations.

Typing judgements will take the form $\Omega \vdash \underline{e} : \delta$ and the straightforward typing rules are given in figure A.9 in appendix A.

We will let θ range over type substitutions mapping type variables to types. We will write $\delta\theta$ for application of a substitution to a type. Similarly we will also apply substitutions to contexts and explicitly typed terms. We say that $\Omega \vdash \underline{e} : \delta$ can be instantiated to $\Omega' \vdash \underline{e}' : \delta'$ by θ (written $\Omega \vdash \underline{e} : \delta \lesssim_\theta \Omega' \vdash \underline{e}' : \delta'$) iff $\Omega\theta \subseteq \Omega'$, $\underline{e}\theta \equiv \underline{e}'$ and $\delta\theta \equiv \delta'$. We will also write $\Omega \vdash \underline{e} : \delta \lesssim \Omega' \vdash \underline{e}' : \delta'$ iff there exists a type substitution θ such that $\Omega \vdash \underline{e} : \delta \lesssim_\theta \Omega' \vdash \underline{e}' : \delta'$. We will say that a typing judgement $\Omega \vdash \underline{e} : \delta$ is a principal typing of e iff $\lfloor \underline{e} \rfloor \equiv e$ and $\Omega \vdash \underline{e} : \delta \lesssim \Omega' \vdash \underline{e}' : \delta'$. for any $\Omega' \vdash \underline{e}' : \delta'$ such that $\lfloor \underline{e}' \rfloor \equiv e$. The following result is due to Damas and Milner [DM82].

Theorem 5.3.1 (Principal typings)
There exists an algorithm `W` such that `W`(e) succeeds if and only if e is well-typed and yields $(\Omega, \underline{e}, \delta)$ such that $\Omega \vdash \underline{e} : \delta$ is a principal typing judgement for e.

5.4 The modified type system

The type system presented in chapter 3 has an important shortcoming; it does not enjoy a principal typing property. In this section, we will remedy this shortcoming and modify the type system of chapter 3 to allow for a principal typing property.

5.4.1 Annotation language

To achieve a principal typing property we need to extend the language of annotations to include annotation variables as follows.

$$\begin{array}{rcl} \kappa & ::= & \checkmark \mid ! \mid k \\ \iota & ::= & [\eta, \xi] \\ \eta & ::= & n \mid \omega \mid i \mid \eta \oplus \kappa \\ \xi & ::= & n \mid \omega \mid j \mid \xi \oplus \kappa \end{array}$$

Thus, we can now annotate bindings (and binding types) with a $\checkmark$, an ! or an annotation variable k. Similarly η now ranges over natural numbers, ω and annotation variables i. However we also introduce an annotation of the form $\eta \oplus \kappa$ which might need some explanation. Consider the two typing rules for bindings.

$$\text{Bind-}\checkmark \frac{\Gamma \vdash e \rightsquigarrow \tilde{e} : \tau}{\Gamma \vdash x = e \rightsquigarrow x =^{\checkmark} \tilde{e} : (x : \tau_{\checkmark})}$$

$$\text{Bind-}! \frac{\Gamma \vdash e \rightsquigarrow \tilde{e} : \rho^{[\eta+1, \xi+1]}}{\Gamma \vdash x = e \rightsquigarrow x =^{!} \tilde{e} : (x : \rho_{\kappa}^{[\eta,\xi]})}$$

Having two rules for bindings was convenient when we presented the type system in chapter 3 but now it poses a problem. Since we allow bindings to be annotated with an annotation variable k as well we should require that the rules can take care of this case also. This requires us to first combine the two rules into one which could then be generalised. If we interpret $\eta \oplus \checkmark$ as η, $\eta \oplus !$ as $\eta + 1$, $\xi \oplus \checkmark$ as ξ and $\xi \oplus !$ as $\xi + 1$ we could combine the two rules as follows.

$$\text{Binding} \frac{\Gamma \vdash e \rightsquigarrow \tilde{e} : \rho^{[\eta \oplus \kappa, \xi \oplus \kappa]}}{\Gamma \vdash x = e \rightsquigarrow x =^{\kappa} \tilde{e} : x : \rho_{\kappa'}^{[\eta,\xi]}} \quad \kappa' \leq \kappa$$

Now, in order to generalise this rule to handle annotation variables as well we introduce the annotations $\eta \oplus \kappa$ and $\xi \oplus \kappa$. Clearly this means that structurally different annotations may mean the same thing. For example the annotations 2, $1 \oplus !$ and $2 \oplus \checkmark$ all mean the same thing and we therefore identify them. That is we define two equivalences as the congruent closures of $\eta \oplus \checkmark \equiv \eta$ and $\xi \oplus \checkmark \equiv \xi$ respectively. Note that this means that annotations can "change shape" when a substitution is applied to them. For example, $i \oplus k[i{:=}1, k{:=}!] \equiv 1 \oplus ! \equiv 2$. Finally note that a closed annotation (that is an annotation not containing annotation variables) can be considered as an annotation in the annotation language defined in chapter 2.

5.4.2 Type language

We will also modify the type language of chapter 3 in two respects. First the annotations in the types will be of the form defined above which includes annotation variables. Second our type language will also contain bare type variables. There will be a countable infinite number of bare type variables associated with every type variable in the underlying type language. We will write a_n and b_n for bare type variables associated with a and b respectively. The obtained type language is given below.

Bare type variables	a_n, b_n, c_n		
Bare types	ρ	$::=$	$a_n \mid \mathtt{Int} \mid \sigma \to \tau \mid \mathtt{List}\ \sigma\ \kappa\ \iota$
Types	τ	$::=$	ρ^{ι}
Binding types	σ	$::=$	τ_{κ}

We will write $\lceil\sigma\rceil$, $\lceil\tau\rceil$ and $\lceil\rho\rceil$, to denote the type expressions (in the underlying type language) obtained by removing annotations. For example $\lceil a_0{}^{[0,0]}_{\checkmark} \to a_1{}^{[0,0]}\rceil \equiv a \to a$. Similarly, we will also write $\lceil\Delta\rceil$ and $\lceil\underline{\tilde{e}}\rceil$ for the context and term obtained accordingly.

We will let $\tilde{\theta}$ range over type substitutions mapping bare type variables to bare types and annotation variables to annotations. We will require that $\lceil\tilde{\theta}(a_n)\rceil = \lceil\tilde{\theta}(a_m)\rceil$, that is substitutions must map bare type variables associated with the same type variable (in the underlying type system) to bare types with the same underlying structure. We will say that a type substitution $\tilde{\theta}$ is closing if $\tau\tilde{\theta}$ is closed (ie do not contain any bare type variables or annotation variables) for any type τ. We will let $\tilde{\vartheta}$ range over closing substitutions. Finally note that a closed type expression can be considered as a type expression in the type language of chapter 3. This is indeed the reason why we need the notion of closing substitutions.

5.4.3 Term language

We will modify the annotated term languages in two respects. First the annotations will be of the form defined above. Second, we will make also the annotated term languages explicitly typed.

Variables	x, y, z		
Values	$\underline{\tilde{v}}$	$::=$	$\lambda x.\underline{\tilde{e}} \mid n \mid \mathtt{nil} \mid \mathtt{cons}\ x\ y$
Expressions	$\underline{\tilde{e}}$	$::=$	$\underline{\tilde{v}}^{\iota} \mid x \mid \underline{\tilde{e}}\ x \mid \underline{\tilde{e}}_0 +^{\iota} \underline{\tilde{e}}_1 \mid$
			$\mathtt{let}\ \underline{\tilde{d}} : \Delta\ \mathtt{in}\ \underline{\tilde{e}} \mid$
			$\mathtt{case}\ \underline{\tilde{e}} : \rho\ \mathtt{of}\ \underline{\tilde{alts}}$
Declarations	$\underline{\tilde{d}}$	$::=$	$\epsilon \mid \underline{\tilde{d}}, \underline{\tilde{b}}$
Bindings	$\underline{\tilde{b}}$	$::=$	$x =^{\kappa} \underline{\tilde{e}}$
Alternatives	$\underline{\tilde{alts}}$	$::=$	$\{\mathtt{nil} \Rightarrow \underline{\tilde{e}}_0; \mathtt{cons}\ x\ y \Rightarrow \underline{\tilde{e}}_1\}$

As in the case of the ordinary term language we annotate `let`-expressions with a context giving the types of the bindings and we annotate `case`-expressions

$$\tilde{\vartheta} \models \texttt{if}\ \xi > 0\ \texttt{then}\ \kappa = !\ \text{iff if}\ \xi\tilde{\vartheta} > 0\ \text{then}\ \kappa\tilde{\vartheta} = !$$
$$\tilde{\vartheta} \models \kappa = !\ \text{iff}\ \kappa\tilde{\vartheta} = !$$
$$\tilde{\vartheta} \models \eta \leq \eta'\ \text{iff}\ \eta\tilde{\vartheta} \leq \eta'\tilde{\vartheta}$$
$$\tilde{\vartheta} \models \xi \leq \xi'\ \text{iff}\ \xi\tilde{\vartheta} \leq \xi'\tilde{\vartheta}$$
$$\tilde{\vartheta} \models \kappa \leq \kappa'\ \text{iff}\ \kappa\tilde{\vartheta} \leq \kappa'\tilde{\vartheta}$$
$$\tilde{\vartheta} \models a_i \leq a_j\ \text{iff}\ a_i\tilde{\vartheta} \leq a_j\tilde{\vartheta}$$

Figure 5.1: Definition of $\tilde{\vartheta} \models \pi$

with a ρ saying that the type of the scrutinised term is $\rho^{[0,0]}$. Note that we will let $\underline{\tilde{e}}$ range over explicitly typed annotated terms to avoid confusion with the untyped annotated terms introduced in chapter 2 which are ranged over by $\tilde{e}$. We will again write $\lfloor\underline{\tilde{e}}\rfloor$ for the untyped term obtained by removing type annotations. We will also write $\lceil\underline{\tilde{e}}\rceil$ for the explicitly typed term obtained by removing the annotations from the term (which includes removing annotations from the annotated types in the term).

5.4.4 Constraints

The fact that we have introduced annotation variables and type variables makes some of the side conditions in our typing rules meaningless. For example the side condition if $\xi > 0$ then $\Gamma \rightarrow^* \Gamma, \Gamma$ in the rule Value does not make sense if ξ is an annotation variable. Instead we will modify our typing judgements such that they include a set of constraints. Then when these constraints are satisfied there exist a corresponding typing judgement in the type system of chapter 3. We will have two form of constraints; atomic constraints that will show up in the typing judgements and composite constraints which will show up in the typing rules. Let us first introduce atomic constraints ranged over by π.

$$\begin{array}{lrl} \text{Atomic constraints} & \pi ::= & \texttt{if}\ \xi > 0\ \texttt{then}\ \kappa = ! \mid \kappa = ! \mid \\ & & \eta_0 \leq \eta_1 \mid \xi_0 \leq \xi_1 \mid \kappa_0 \leq \kappa_1 \mid a_n \leq a_m \end{array}$$

We will give a meaning to these atomic constraints by means of a relation $\tilde{\vartheta} \models \pi$ saying that $\tilde{\vartheta}$ models (ie satisfies) π. The relation is defined in figure 5.1 and should be self explanatory. We will let Π range over sets of atomic constraints and we will write $\tilde{\vartheta} \models \Pi$ iff $\tilde{\vartheta}$ models all the constraints in Π. We will also write $\models \Pi$ iff all closing substitutions models Π.

We will now go on and introduce composite constraints which will be the constraints that show up in our typing rules. We will let γ range over composite

$$\frac{}{\Pi \vdash \mathtt{if}\ \xi > 0\ \mathtt{then}\ \emptyset \to^* \emptyset, \emptyset}$$

$$\frac{\Pi \vdash \mathtt{if}\ \xi > 0\ \mathtt{then}\ \Gamma \to^* \Gamma, \Gamma}{\Pi \vdash \mathtt{if}\ \xi > 0\ \mathtt{then}\ \Gamma, x : \tau_\kappa \to^* \Gamma, x : \tau_\kappa, \Gamma, x : \tau_\kappa} \quad \mathtt{if}\ \xi > 0\ \mathtt{then}\ \kappa = ! \in \Pi$$

$$\frac{}{\Pi \vdash \Gamma,\ x : \tau_\kappa \to \Gamma,\ x : \tau_\kappa,\ x : \tau_\kappa} \quad \kappa = ! \in \Pi \qquad \frac{}{\Pi \vdash \Gamma,\ x : \sigma \to \Gamma}$$

$$\frac{}{\Pi \vdash \Gamma \to^* \Gamma} \qquad \frac{\Pi \vdash \Gamma \to^* \Gamma' \quad \Pi \vdash \Gamma' \to \Gamma''}{\Pi \vdash \Gamma \to^* \Gamma''} \qquad \frac{\Pi \vdash \Delta \to^* \Gamma}{\Pi \vdash \Delta \vdash \Gamma}$$

$$\frac{}{\Pi \vdash [\eta, \xi] \leq [\eta', \xi']} \quad \eta' \leq \eta, \xi \leq \xi' \in \Pi$$

$$\frac{}{\Pi \vdash a_n \leq a_m} \quad a_n \leq a_m \in \Pi \qquad \frac{}{\Pi \vdash \mathtt{Int} \leq \mathtt{Int}}$$

$$\frac{\Pi \vdash \sigma' \leq \sigma \qquad \Pi \vdash \tau \leq \tau'}{\Pi \vdash \sigma \to \tau \leq \sigma' \to \tau'} \qquad \frac{\Pi \vdash \sigma \leq \sigma' \qquad \Pi \vdash \iota' \leq \iota}{\Pi \vdash \mathtt{List}\ \sigma\ \kappa\ \iota \leq \mathtt{List}\ \sigma'\ \kappa'\ \iota'} \quad \kappa' \leq \kappa \in \Pi$$

$$\frac{\Pi \vdash \tau \leq \tau'}{\Pi \vdash \tau_\kappa \leq \tau'_{\kappa'}} \quad \kappa' \leq \kappa \in \Pi \qquad \frac{\Pi \vdash \rho \leq \rho' \qquad \Pi \vdash \iota' \leq \iota}{\Pi \vdash \rho^\iota \leq \rho'^{\iota'}}$$

Figure 5.2: Constraint derivation

constraints.

$$\begin{array}{rrl} \text{Composite constraints} & \gamma ::= & \mathtt{if}\ \xi > 0\ \mathtt{then}\ \Gamma \to^* \Gamma, \Gamma \mid \Gamma \to \Gamma' \mid \Gamma \to^* \Gamma' \mid \\ & & \Delta \vdash \Gamma \mid \iota_0 \leq \iota_1 \mid \rho_0 \leq \rho_1 \mid \tau_0 \leq \tau_1 \mid \sigma_0 \leq \sigma_1 \end{array}$$

Note that our notation is highly ambiguous. For example we now have both a constraint $\tau_0 \leq \tau_1$ which is pure syntax and a statement $\tau_0 \leq \tau_1$ in our meta language. If a constraint γ is closed we will let $[\![\gamma]\!]$ denote this corresponding statement in the meta language. We will give meaning to composite constraints by defining when a composite constraint γ can be derived from a set of atomic constraints Π (written $\Pi \vdash \gamma$). The definition of $\Pi \vdash \gamma$ appears in figure 5.2. We will frequently use the following result.

Lemma 5.4.1 (Weakening)
If $\Pi \vdash \gamma$ then $\Pi, \Pi' \vdash \gamma$.

Proof 5.4.2
By induction over $\Pi \vdash \gamma$.

We will say that a closing substitution $\tilde{\vartheta}$ models γ (written $\tilde{\vartheta} \models \gamma$) iff there exists a set of atomic constraints Π such that $\tilde{\vartheta} \models \Pi$ and $\Pi \vdash \gamma$. We will write $\models \gamma$ iff $\tilde{\vartheta} \models \gamma$ for all $\tilde{\vartheta}$.

We also need to define what it means to apply a substitution $\tilde{\theta}$ to an atomic constraint π. If π is not of the form $a_n \leq a_m$ we simply apply the substitution to the components of the constraint. However if $\pi \equiv a_n \leq a_m$ then $a_n\tilde{\theta} \leq a_m\tilde{\theta}$ is not necessarily an atomic constraint. However we require from $\tilde{\theta}$ that $\lceil\tilde{\theta}(a_i)\rceil \equiv \lceil\tilde{\theta}(a_j)\rceil$. This means that we can break $a_n\tilde{\theta} \leq a_m\tilde{\theta}$ down into a set of atomic constraints Π such that $\Pi \vdash a_n\tilde{\theta} \leq a_m\tilde{\theta}$. The definition is straightforward and has been omitted. For composite constraints the situation is simpler and we simply apply the substitution to the components of the constraint.

Now we are ready to state the relationship between our composite constraints and the corresponding statement in the meta language by means of a soundness and a completeness lemma.

Lemma 5.4.3 (Soundness of $\tilde{\vartheta} \models \gamma$)
If $\tilde{\vartheta} \models \gamma$ then $[\![\gamma\tilde{\vartheta}]\!]$.

Proof 5.4.4
By induction over the size of γ.

Lemma 5.4.5 (Completeness of $\tilde{\vartheta} \models \gamma$)
If $[\![\gamma]\!]$ then $\models \gamma$.

Proof 5.4.6
By induction over the size of γ.

5.4.5 Typing judgements

Typing judgements for expression now take the form $\Pi\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau$ which should be read as "from the atomic constraints Π we can derive that in the context Γ, the expression e can be annotated as $\tilde{e}$ having type τ". We will also modify our typing judgements for values, alternatives, bindings and declarations in the same way.

The typing rules are directly derived from the typing rules in chapter 3 and appear in appendix A. We will frequently use the following lemma.

Lemma 5.4.7 (Weakening)
If $\Pi\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau$ then $\Pi, \Pi'\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau$

Proof 5.4.8
By induction over the size of $\underline{e}$ using the weakening lemma for $\Pi \vdash \gamma$.

We will write $\tilde{\vartheta} \models \Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau$ iff there exists Π such that $\tilde{\vartheta} \models \Pi$ and $\Pi\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau$. We will also write $\models \Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau$ iff $\tilde{\vartheta} \models \Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau$ for all $\tilde{\vartheta}$. Using this notation we can state the soundness and completeness of the modified type system with respect to the type system of chapter 3.

Proposition 5.4.9 (Soundness)
If $\tilde{\vartheta} \models \Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau$ then $\Gamma\tilde{\vartheta} \vdash \lfloor \underline{e} \rfloor \rightsquigarrow \lfloor \underline{\tilde{e}}\tilde{\vartheta} \rfloor : \tau\tilde{\vartheta}$

Proof 5.4.10
By induction over $\underline{e}$ using the soundness of $\tilde{\vartheta} \models \gamma$.

Proposition 5.4.11 (Completeness)
If $\Gamma \vdash e \rightsquigarrow \tilde{e} : \tau$ then $\models \Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau$ where $\lfloor \underline{e} \rfloor \equiv e$, $\lfloor \underline{\tilde{e}} \rfloor \equiv \tilde{e}$.

Proof 5.4.12
By induction over e using the completeness of $\tilde{\vartheta} \models \gamma$ and the weakening lemma.

5.5 Computing principal typings

Up to to this point we have introduced the modified type system and argued its correspondence with the type system of chapter 3. Now we will go on and show how we can compute a principal typing judgement in the modified type system. We will also show that the generated constraints are in a restricted form for which there exists an algorithm that efficiently can find an optimal solution to the constraints.

The algorithm will take a term e as its input. It will first compute, using $\mathtt{W}(e)$, a triple $(\Omega, \underline{e}, \delta)$ such that $\Omega \vdash \underline{e} : \delta$ is a principal typing for e. The next stage is to decorate the triple with fresh annotation variables so we obtain a triple $(\Delta, \underline{\tilde{e}}, \tau)$. Then we finally compute a constraint set Π and a context Γ such that $\Pi\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau$ and $\Pi \vdash \Delta \vdash \Gamma$.

5.5.1 Decorating

The algorithm $\mathtt{decorate}(\Omega, \underline{e}, \delta)$ that decorates the triple $(\Omega, \underline{e}, \delta)$ is entirely straightforward and it has been omitted. We will however need the following properties of the algorithm.

Lemma 5.5.1

- $\mathtt{decorate}(\Omega, \underline{e}, \delta)$ always succeeds.
- $\lceil \mathtt{decorate}(\Omega, \underline{e}, \delta) \rceil \equiv (\Omega, \underline{e}, \delta)$
- If $(\Omega, \underline{e}, \delta) \lesssim \lceil (\Delta, \underline{\tilde{e}}, \tau) \rceil$ then $\mathtt{decorate}(\Omega, \underline{e}, \delta) \lesssim (\Delta, \underline{\tilde{e}}, \tau)$.
- The annotations in $\mathtt{decorate}(\Omega, \underline{e}, \delta)$ are distinct.

$\Vdash$ if $\xi > 0$ then $\Gamma' \rightarrow^* \Gamma', \Gamma'$
$\Vdash \Gamma \rightarrow \Gamma'$ iff if $x : \sigma \in \Gamma'$ then $x : \sigma \in \Gamma$
$\Vdash \Gamma \rightarrow^* \Gamma'$ iff if $x : \sigma \in \Gamma'$ then $x : \sigma \in \Gamma$
$\Vdash \Delta \vdash \Gamma$ iff if $x : \sigma \in \Gamma$ then $x : \sigma \in \Delta$
$\Vdash \rho_0 \leq \rho_1$ iff $\lceil \rho_0 \rceil \equiv \lceil \rho_1 \rceil$
$\Vdash \tau_0 \leq \tau_1$ iff $\lceil \tau_0 \rceil \equiv \lceil \tau_1 \rceil$
$\Vdash \sigma_0 \leq \sigma_1$ iff $\lceil \sigma_0 \rceil \equiv \lceil \sigma_1 \rceil$
$\Vdash \iota_0 \leq \iota_1$

Figure 5.3: Definition of $\Vdash \gamma$

Proof 5.5.2
The proof follows the structure of the algorithm and is completely straightforward.

5.5.2 Inferring constraint sets

After we have decorated $(\Omega, \underline{e}, \delta)$ as $(\Delta, \tilde{\underline{e}}, \tau)$ the next step is to compute a constraint set Π and a context Γ such that $\Pi \,;\, \Gamma \vdash \underline{e} \rightsquigarrow \tilde{\underline{e}} : \tau$ and $\Pi \vdash \Delta \vdash \Gamma$. The derivation of $\Pi \,;\, \Gamma \vdash \underline{e} \rightsquigarrow \tilde{\underline{e}} : \tau$ depends on a number of derivations of composite constraints. Consequently the algorithm depends on an algorithm, which we will call `infer`, that given a constraint γ can compute a Π such that $\Pi \vdash \gamma$. It is however not always possible to find such a Π which means that `infer` might fail. For example if $\gamma \equiv \texttt{Int} \leq \sigma \rightarrow \tau$ then there is no Π such that $\Pi \vdash \gamma$. We therefore introduce a relation $\Vdash \gamma$ (which should be read as "γ is derivable") which we will be a sufficient (and necessary) condition for `infer` to succeed. The definition of $\Vdash \gamma$ is given in figure 5.3.

We also need to show that the constraints generated by `infer` are of a specific form. For this purpose we use the notion of covariance and contravariance. We will write $\ulcorner \rho \urcorner^+$ for the set of annotations of ρ that occur in covariant positions and $\ulcorner \rho \urcorner^-$ for those occurring in contravariant positions. Similarly we will also define $\ulcorner \tau \urcorner^+$, $\ulcorner \tau \urcorner^-$, $\ulcorner \sigma \urcorner^+$ and $\ulcorner \sigma \urcorner^-$. This is done in figure 5.4. We also extend the notion of covariance and contravariance to atomic constraints in figure 5.5 and to composite constraints in figure 5.6. The definition of $\texttt{infer}(\gamma)$ is straightforward and has been omitted. The following result states the soundness of $\Vdash \gamma$.

Lemma 5.5.3 (Soundness of $\Vdash \gamma$)
If $\Vdash \gamma$ then $\texttt{infer}(\gamma)$ succeeds yielding Π such that

- $\Pi \vdash \gamma$
- If $\Pi' \vdash \gamma\tilde{\theta}$ then $\Pi\tilde{\theta} \subseteq \Pi'$.

$$
\begin{array}{rlrl}
\ulcorner[\eta,\xi]\urcorner^{+} &= \{\xi\} & \ulcorner[\eta,\xi]\urcorner^{-} &= \{\eta\} \\
\ulcorner a_m \urcorner^{+} &= \emptyset & \ulcorner a_m \urcorner^{-} &= \emptyset \\
\ulcorner \mathtt{Int} \urcorner^{+} &= \emptyset & \ulcorner \mathtt{Int} \urcorner^{-} &= \emptyset \\
\ulcorner \sigma \to \tau \urcorner^{+} &= \ulcorner \sigma \urcorner^{-} \cup \ulcorner \tau \urcorner^{+} & \ulcorner \sigma \to \tau \urcorner^{-} &= \ulcorner \sigma \urcorner^{+} \cup \ulcorner \tau \urcorner^{-} \\
\ulcorner \mathtt{List}\, \sigma\, \kappa\, \iota \urcorner^{+} &= \ulcorner \sigma \urcorner^{+} \cup \ulcorner \iota \urcorner^{-} & \ulcorner \mathtt{List}\, \sigma\, \kappa\, \iota \urcorner^{-} &= \ulcorner \sigma \urcorner^{-} \cup \{\kappa\} \cup \ulcorner \iota \urcorner^{+} \\
\ulcorner \rho^{\iota} \urcorner^{+} &= \ulcorner \rho \urcorner^{+} \cup \ulcorner \iota \urcorner^{-} & \ulcorner \rho^{\iota} \urcorner^{-} &= \ulcorner \rho \urcorner^{-} \cup \ulcorner \iota \urcorner^{+} \\
\ulcorner \tau_{\kappa} \urcorner^{+} &= \ulcorner \tau \urcorner^{+} & \ulcorner \tau_{\kappa} \urcorner^{-} &= \ulcorner \tau \urcorner^{-} \cup \{\kappa\}
\end{array}
$$

Figure 5.4: Covariance and contravariance for types

$$
\begin{array}{rlrl}
\ulcorner \mathtt{if}\ \xi > 0\ \mathtt{then}\ \kappa = !\urcorner^{+} &= \{\kappa\} & \ulcorner \mathtt{if}\ \xi > 0\ \mathtt{then}\ \kappa = !\urcorner^{-} &= \{\xi\} \\
\ulcorner \kappa = !\urcorner^{+} &= \{\kappa\} & \ulcorner \kappa = !\urcorner^{-} &= \emptyset \\
\ulcorner \eta_0 \le \eta_1 \urcorner^{+} &= \{\eta_1\} & \ulcorner \eta_0 \le \eta_1 \urcorner^{-} &= \{\eta_0\} \\
\ulcorner \xi_0 \le \xi_1 \urcorner^{+} &= \{\xi_1\} & \ulcorner \xi_0 \le \xi_1 \urcorner^{-} &= \{\xi_0\} \\
\ulcorner \kappa_0 \le \kappa_1 \urcorner^{+} &= \{\kappa_1\} & \ulcorner \kappa_0 \le \kappa_1 \urcorner^{-} &= \{\kappa_0\}
\end{array}
$$

Figure 5.5: Covariance and contravariance for atomic constraints

- $\ulcorner \Pi \urcorner^{+} \subseteq \ulcorner \gamma \urcorner^{+}$
- $\ulcorner \Pi \urcorner^{-} \subseteq \ulcorner \gamma \urcorner^{-}$

Proof 5.5.4
By induction over the size of γ.

We are now ready to define a function $\mathtt{infer}(\Delta, \underline{\tilde{e}}, \tau)$ that infers a Π and a Γ such that $\Pi\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau$. We will actually define a set of mutually recursive functions, one function for each form of judgement we have. The functions are fully defined in appendix B and follow the typing rules closely.

The definition of $\mathtt{infer}(\Delta, \underline{\tilde{e}}, \tau)$ depends on $\mathtt{infer}(\gamma)$ which can fail and thus $\mathtt{infer}(\Delta, \underline{\tilde{e}}, \tau)$ can also fail. We therefore define a relation $\Vdash (\Delta, \underline{\tilde{e}}, \tau)$ as $\Vdash (\Delta, \underline{\tilde{e}}, \tau)$ iff $\lceil \Delta \rceil \vdash \lceil \underline{\tilde{e}} \rceil : \lceil \tau \rceil$. Knowing that $\Vdash (\Delta, \underline{\tilde{e}}, \tau)$ will be sufficient to guarantee that $\mathtt{infer}(\Delta, \underline{\tilde{e}}, \tau)$ succeeds.

We define $\ulcorner \tilde{e} \urcorner$ to be the annotations in $\tilde{e}$ and $\ulcorner \underline{\tilde{e}} \urcorner = \ulcorner \lfloor \underline{\tilde{e}} \rfloor \urcorner$. We also extend the notion of covariance and contravariance to triples of the form $(\Delta, \underline{\tilde{e}}, \tau)$ in figure 5.7. We are then finally ready to state the soundness of $\Vdash (\Delta, \underline{\tilde{e}}, \tau)$.

$$\ulcorner \texttt{if}\ \xi > 0\ \texttt{then}\ \Gamma \to^* \Gamma, \Gamma \urcorner^+ = \emptyset$$
$$\ulcorner \texttt{if}\ \xi > 0\ \texttt{then}\ \Gamma \to^* \Gamma, \Gamma \urcorner^- = \{\xi\} \cup \ulcorner \Gamma \urcorner^-$$

$$\begin{array}{rlrl}
\ulcorner \Gamma \to \Gamma' \urcorner^+ &= \emptyset & \ulcorner \Gamma \to \Gamma' \urcorner^- &= \ulcorner \Gamma \urcorner^- \\
\ulcorner \Gamma \to^* \Gamma' \urcorner^+ &= \emptyset & \ulcorner \Gamma \to^* \Gamma' \urcorner^- &= \ulcorner \Gamma \urcorner^- \\
\ulcorner \Delta \vdash \Gamma \urcorner^+ &= \emptyset & \ulcorner \Delta \vdash \Gamma \urcorner^- &= \ulcorner \Delta \urcorner^- \\
\ulcorner \iota_0 \leq \iota_1 \urcorner^+ &= \ulcorner \iota_0 \urcorner^- \cup \ulcorner \iota_1 \urcorner^+ & \ulcorner \iota_0 \leq \iota_1 \urcorner^- &= \ulcorner \iota_0 \urcorner^+ \cup \ulcorner \iota_1 \urcorner^- \\
\ulcorner \rho_0 \leq \rho_1 \urcorner^+ &= \ulcorner \rho_0 \urcorner^- \cup \ulcorner \rho_1 \urcorner^+ & \ulcorner \rho_0 \leq \rho_1 \urcorner^- &= \ulcorner \rho_0 \urcorner^+ \cup \ulcorner \rho_1 \urcorner^- \\
\ulcorner \tau_0 \leq \tau_1 \urcorner^+ &= \ulcorner \tau_0 \urcorner^- \cup \ulcorner \tau_1 \urcorner^+ & \ulcorner \tau_0 \leq \tau_1 \urcorner^- &= \ulcorner \tau_0 \urcorner^+ \cup \ulcorner \tau_1 \urcorner^- \\
\ulcorner \sigma_0 \leq \sigma_1 \urcorner^+ &= \ulcorner \sigma_0 \urcorner^- \cup \ulcorner \sigma_1 \urcorner^+ & \ulcorner \sigma_0 \leq \sigma_1 \urcorner^- &= \ulcorner \sigma_0 \urcorner^+ \cup \ulcorner \sigma_1 \urcorner^-
\end{array}$$

Figure 5.6: Covariance and contravariance for constraints

$$\begin{aligned}
\ulcorner (\Delta, \underline{\tilde{e}}, \tau) \urcorner^+ &= \ulcorner \Delta \urcorner^- \cup \ulcorner \underline{\tilde{e}} \urcorner^+ \cup \ulcorner \tau \urcorner^+ \\
\ulcorner (\Delta, \underline{\tilde{e}}, \tau) \urcorner^- &= \ulcorner \Delta \urcorner^+ \cup \ulcorner \underline{\tilde{e}} \urcorner^- \cup \ulcorner \tau \urcorner^- \\
\ulcorner \underline{\tilde{e}} \urcorner^+ &= \{\xi \mid \xi \in \ulcorner \underline{\tilde{e}} \urcorner\} \cup \{\kappa \mid \kappa \in \ulcorner \underline{\tilde{e}} \urcorner\} \cup \ulcorner \mathrm{types}(\underline{\tilde{e}}) \urcorner^+ \cup \ulcorner \mathrm{types}(\underline{\tilde{e}}) \urcorner^- \\
\ulcorner \underline{\tilde{e}} \urcorner^- &= \{\eta \mid \eta \in \ulcorner \underline{\tilde{e}} \urcorner\} \cup \{\kappa \mid \kappa \in \ulcorner \underline{\tilde{e}} \urcorner\} \cup \ulcorner \mathrm{types}(\underline{\tilde{e}}) \urcorner^+ \cup \ulcorner \mathrm{types}(\underline{\tilde{e}}) \urcorner^-
\end{aligned}$$

Figure 5.7: Covariance and contravariance for $(\Delta, \underline{\tilde{e}}, \tau)$

Proposition 5.5.5 (Soundness of $\Vdash (\Delta, \underline{\tilde{e}}, \tau)$)
If $\Vdash (\Delta, \underline{\tilde{e}}, \tau)$ then $\texttt{infer}(\Delta, \underline{\tilde{e}}, \tau)$ will succeed yielding (Π, Γ) such that

- $\Pi\,;\ \Gamma \vdash \lceil \underline{\tilde{e}} \rceil \rightsquigarrow \underline{\tilde{e}} : \tau$.
- $\Vdash \Delta \vdash \Gamma$
- If $\Pi'; \Gamma' \vdash \underline{e}' \rightsquigarrow \underline{\tilde{e}}' : \tau'$, $\Vdash \Delta' \vdash \Gamma'$ and $(\Delta, \underline{\tilde{e}}, \tau) \lesssim_{\tilde{\theta}} (\Delta', \underline{\tilde{e}}', \tau')$ then $\Pi\tilde{\theta} \subseteq \Pi'$ and $\Gamma\tilde{\theta} \subseteq \Gamma'$.
- $\ulcorner \Pi \urcorner^+ \subseteq \ulcorner (\Delta, \underline{\tilde{e}}, \tau) \urcorner^+ \cup \{0_\eta\} \cup \{\eta \oplus \kappa \mid \eta \in \ulcorner (\Delta, \underline{\tilde{e}}, \tau) \urcorner^+, \kappa \in \ulcorner \underline{\tilde{e}} \urcorner\}$
- $\ulcorner \Pi \urcorner^- \subseteq \ulcorner (\Delta, \underline{\tilde{e}}, \tau) \urcorner^- \cup \{0_\xi\} \cup \{\xi \oplus \kappa \mid \xi \in \ulcorner (\Delta, \underline{\tilde{e}}, \tau) \urcorner^-, \kappa \in \ulcorner \underline{\tilde{e}} \urcorner\}$

Proof 5.5.6
By induction over the size of $\underline{\tilde{e}}$ using the soundness of $\Vdash \gamma$.

5.5.3 Principal typings

We can finally put the pieces together and define a function $\texttt{principal}(e)$ which computes a principal typing. The definition follows below.

$$\begin{array}{rll} \texttt{principal}(e) = & (\Pi, \Delta, \Gamma, \underline{e}, \underline{\tilde{e}}, \tau) & \\ & \text{where } (\Omega, \underline{e}, \delta) = \texttt{W}(e) & \\ & \quad (\Delta, \underline{\tilde{e}}, \tau) = \texttt{decorate}(\Omega, \underline{e}, \tau) & \\ & \quad (\Pi_0, \Gamma) = \texttt{infer}(\Delta, \underline{\tilde{e}}, \tau) & \\ & \quad \Pi_1 = \texttt{infer}(\Delta \vdash \Gamma) & \\ & \quad \Pi = \Pi_0, \Pi_1 & \end{array}$$

To prove that $\texttt{principal}(e)$ actually computes a principal typing we need the following completeness results for $\Vdash \gamma$ and $\Vdash (\Delta, \underline{\tilde{e}}, \tau)$.

Lemma 5.5.7 (Completeness of $\Vdash \gamma$)
If $\Pi \vdash \gamma$ then $\Vdash \gamma$.

Proof 5.5.8
By induction over $\Pi \vdash \gamma$.

Lemma 5.5.9 (Completeness of $\Vdash (\Delta, \underline{\tilde{e}}, \tau)$)
If $\Pi\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau$ and $\Vdash \Delta \vdash \Gamma$ then $\Vdash (\Delta, \underline{\tilde{e}}, \tau)$

Proof 5.5.10
By induction over the size of $\underline{e}$ using the completeness of $\Vdash \gamma$.

We need to prove that the constraints computed by $\texttt{principal}(e)$ are of a restricted form. Let ϕ range over constraints of the following form.

$$\begin{array}{rcl} \zeta & ::= & i \mid 0 \mid i \oplus k \\ \chi & ::= & j \mid 0 \mid j \oplus k \\ \phi & ::= & \texttt{if}\ \chi > 0\ \texttt{then}\ k = ! \mid k = ! \mid i_0 \leq \zeta \mid \chi \leq j_1 \mid k_0 \leq k_1 \mid a_i \leq a_j \end{array}$$

We will take Φ to range over sets of restricted constraints.

We will write $\Pi\,;\,\Delta\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau$ iff $\Pi\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau$ and $\Pi \vdash \Delta \vdash \Gamma$. We will let $\Pi\,;\,\Delta\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau \lesssim_{\tilde{\theta}} \Pi'\,;\,\Delta'\,;\,\Gamma' \vdash \underline{e}' \rightsquigarrow \underline{\tilde{e}}' : \tau'$ iff $\Pi\tilde{\theta} \subseteq \Pi'$, $\Delta\tilde{\theta} \subseteq \Delta'$, $\Gamma\tilde{\theta} \subseteq \Gamma'$, $\underline{\tilde{e}}\tilde{\theta} \equiv \underline{\tilde{e}}'$ and $\tau\tilde{\theta} \equiv \tau'$. Finally, we say that $\Pi\,;\,\Delta\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau$ is a principal typing judgement for e iff for any other typing $\Pi'\,;\,\Delta'\,;\,\Gamma' \vdash \underline{e}' \rightsquigarrow \underline{\tilde{e}}' : \tau'$ for e it is the case that $\Pi\,;\,\Delta\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau \lesssim \Pi'\,;\,\Delta'\,;\,\Gamma' \vdash \underline{e}' \rightsquigarrow \underline{\tilde{e}}' : \tau'$.

Now we are ready to state the main result.

Theorem 5.5.11 (Principal typings)
If and only if e is welltyped, $\texttt{principal}(e)$ succeeds yielding $(\Pi, \Delta, \Gamma, \underline{e}, \underline{\tilde{e}}, \tau)$ such that

- $\Pi\,;\,\Delta\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau$ is a principal typing for e.

- If $\pi \in \Pi$ then
  - π is in the set of restricted constraints ranged over by ϕ.
  - If $i \in \ulcorner \pi \urcorner^+$ then $i \notin \ulcorner \underline{\tilde{e}} \urcorner$.
  - If $i \oplus k \in \ulcorner \pi \urcorner^+$ then $i \notin \ulcorner \underline{\tilde{e}} \urcorner$ and $k \in \ulcorner \underline{\tilde{e}} \urcorner$.
  - If $j \in \ulcorner \pi \urcorner^-$ then $j \notin \ulcorner \underline{\tilde{e}} \urcorner$.
  - If $j \oplus k \in \ulcorner \pi \urcorner^-$ then $j \notin \ulcorner \underline{\tilde{e}} \urcorner$ and $k \in \ulcorner \underline{\tilde{e}} \urcorner$.

Proof 5.5.12
If e is ill-typed then $\mathtt{W}(e)$ will fail and therefore also $\mathtt{principal}(e)$. Assume instead that e is well-typed. We then know from the principal typing theorem that $\mathtt{W}(e)$ succeeds, yielding $(\Omega, \underline{e}, \delta)$ such that $\Omega \vdash \underline{e} : \delta$ is a principal typing for e. We know that $\mathtt{decorate}(\Omega, \underline{e}, \delta)$ always succeeds yielding $(\Delta, \underline{\tilde{e}}, \tau)$ such that

$$\lceil (\Delta, \underline{\tilde{e}}, \tau) \rceil \equiv (\Omega, \underline{e}, \delta)$$

$$(\Delta, \underline{\tilde{e}}, \tau) \lesssim (\Delta', \underline{\tilde{e}}', \tau') \text{ if } (\Omega, \underline{e}, \delta) \lesssim \lceil (\Delta', \underline{\tilde{e}}', \tau') \rceil.$$

Now from $\Omega \vdash \underline{e} : \delta$ and $\lceil (\Delta, \underline{\tilde{e}}, \tau) \rceil \equiv (\Omega, \underline{e}, \delta)$ we know that

$$\Vdash (\Delta, \underline{\tilde{e}}, \tau).$$

Thus we now by the soundness of $\Vdash (\Delta, \underline{\tilde{e}}, \tau)$ that $\mathtt{infer}(\Delta, \underline{\tilde{e}}, \tau)$ will succeed yielding (Π_0, Γ) such that

$$\Pi_0 \,;\, \Gamma \vdash \lceil \underline{\tilde{e}} \rceil \rightsquigarrow \underline{\tilde{e}} : \tau,$$

$$\Vdash \Delta \vdash \Gamma,$$

$$\begin{array}{l} \text{if } \Pi' \,;\, \Gamma' \vdash \underline{e}' \rightsquigarrow \underline{\tilde{e}}' : \tau', \Vdash \Delta' \vdash \Gamma' \text{ and } (\Delta, \underline{\tilde{e}}, \tau) \lesssim_{\tilde{\theta}} (\Delta', \underline{\tilde{e}}', \tau') \\ \text{then } \Pi_0 \tilde{\theta} \subseteq \Pi' \text{ and } \Gamma \tilde{\theta} \subseteq \Gamma', \end{array}$$

$$\ulcorner \Pi_0 \urcorner^+ \subseteq \ulcorner (\Delta, \underline{\tilde{e}}, \tau) \urcorner^+ \cup \{0_\eta\} \cup \{\eta \oplus \kappa \mid \eta \in \ulcorner (\Delta, \underline{\tilde{e}}, \tau) \urcorner^+, \kappa \in \ulcorner \underline{\tilde{e}} \urcorner\}$$

and

$$\ulcorner \Pi_0 \urcorner^- \subseteq \ulcorner (\Delta, \underline{\tilde{e}}, \tau) \urcorner^- \cup \{0_\xi\} \cup \{\xi \oplus \kappa \mid \xi \in \ulcorner (\Delta, \underline{\tilde{e}}, \tau) \urcorner^-, \kappa \in \ulcorner \underline{\tilde{e}} \urcorner\}.$$

We also know by the soundness of $\Vdash \Delta \vdash \Gamma$ that $\mathtt{infer}(\Delta \vdash \Gamma)$ succeeds yielding Π_1 such that

$$\Pi_1 \vdash \Delta \vdash \Gamma,$$

$$\text{if } \Pi_1' \vdash (\Delta \vdash \Gamma)\tilde{\theta} \text{ then } \Pi_1 \tilde{\theta} \subseteq \Pi_1',$$

$$\ulcorner \Pi_1 \urcorner^+ \subseteq \ulcorner \Delta \vdash \Gamma \urcorner^+$$

and

$$\ulcorner \Pi_1 \urcorner^- \subseteq \ulcorner \Delta \vdash \Gamma \urcorner^-.$$

Let $\Pi = \Pi_0, \Pi_1$. We can now conclude that $\texttt{principal}(e)$ succeeds yielding $(\Pi, \Delta, \Gamma, \underline{e}, \tilde{\underline{e}}, \tau)$. We also need to show that $\Pi\,;\,\Delta\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \tilde{\underline{e}} : \tau$ is a principal typing for e. First we know from $\Pi_0\,;\Gamma \vdash \lceil\underline{\tilde{e}}\rceil \rightsquigarrow \tilde{\underline{e}} : \tau$, $\lceil\underline{\tilde{e}}\rceil \equiv \underline{e}$ and the weakening lemma that

$$\Pi\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \tilde{\underline{e}} : \tau.$$

We also know by $\Pi_1 \vdash \Delta \vdash \Gamma$ and the weakening lemma that

$$\Pi \vdash \Delta \vdash \Gamma$$

so

$$\Pi\,;\,\Delta\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \tilde{\underline{e}} : \tau.$$

Now, if

$$\Pi'\,;\,\Delta'\,;\,\Gamma' \vdash \underline{e}' \rightsquigarrow \tilde{\underline{e}}' : \tau'$$

is a judgement for e, we need to show that $\Pi\,;\,\Delta\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \tilde{\underline{e}} : \tau \lesssim \Pi'\,;\,\Delta'\,;\,\Gamma' \vdash \underline{e}' \rightsquigarrow \tilde{\underline{e}}' : \tau'$. By definition, $\Pi'\,;\,\Delta'\,;\,\Gamma' \vdash \underline{e}' \rightsquigarrow \tilde{\underline{e}}' : \tau'$ means that

$$\Pi'\,;\,\Gamma' \vdash \underline{e}' \rightsquigarrow \tilde{\underline{e}}' : \tau'$$

and

$$\Pi' \vdash \Delta' \vdash \Gamma'.$$

Now, from $\Pi' \vdash \Delta' \vdash \Gamma'$ we know by the completeness of $\Vdash \Delta' \vdash \Gamma'$ that

$$\Vdash \Delta' \vdash \Gamma'$$

so by $\Pi'\,;\,\Gamma' \vdash \underline{e}' \rightsquigarrow \tilde{\underline{e}}' : \tau'$ and the completeness of $\Vdash (\Delta', \tilde{\underline{e}}', \tau')$ we know that

$$\Vdash (\Delta', \tilde{\underline{e}}', \tau')$$

which means that

$$\lceil\Delta'\rceil \vdash \lceil\tilde{\underline{e}}'\rceil : \lceil\tau'\rceil.$$

Now since $\Omega \vdash \underline{e} : \delta$ is a principal judgement we know that

$$\Omega \vdash \underline{e} : \delta \lesssim \lceil\Delta'\rceil \vdash \lceil\tilde{\underline{e}}'\rceil : \lceil\tau'\rceil$$

that is

$$(\Omega, \underline{e}, \delta) \lesssim \lceil(\Delta', \tilde{\underline{e}}', \tau')\rceil.$$

From this we may conclude that there exist $\tilde{\theta}$ such that

$$(\Delta, \tilde{\underline{e}}, \tau) \lesssim_{\tilde{\theta}} (\Delta', \tilde{\underline{e}}', \tau')$$

which implies that

$$\Pi_0\tilde{\theta} \subseteq \Pi'$$

and

$$\Gamma\tilde{\theta} \subseteq \Gamma'.$$

By $(\Delta, \underline{\tilde{e}}, \tau) \lesssim_{\tilde{\theta}} (\Delta', \underline{\tilde{e}}', \tau')$ we also know that

$$\Delta\tilde{\theta} \subseteq \Delta'$$

$$\underline{\tilde{e}}\tilde{\theta} \equiv \underline{\tilde{e}}'$$

$$\tau\tilde{\theta} \equiv \tau'.$$

Now from $\Pi' \vdash \Delta' \vdash \Gamma'$, $\Delta\tilde{\theta} \subseteq \Delta'$ and $\Gamma\tilde{\theta} \subseteq \Gamma'$ we can by a simple inductive argument show that

$$\Pi' \vdash \Delta\tilde{\theta} \vdash \Gamma\tilde{\theta}$$

Thus we may conclude that

$$\Pi_1\tilde{\theta} \subseteq \Pi'.$$

Finally from $\Pi_0\tilde{\theta} \subseteq \Pi'$, $\Pi_1\tilde{\theta} \subseteq \Pi'$ and $\Pi = \Pi_0, \Pi_1$ we know that

$$\Pi\tilde{\theta} \subseteq \Pi'$$

which is what is required for

$$\Pi\,;\,\Delta\,;\,\Gamma \vdash \lceil\underline{\tilde{e}}\rceil \rightsquigarrow \underline{\tilde{e}} : \tau \lesssim \Pi'\,;\,\Delta'\,;\,\Gamma' \vdash \underline{e}' \rightsquigarrow \underline{\tilde{e}}' : \tau'.$$

Thus $\Pi\,;\,\Delta\,;\,\Gamma \vdash \lceil\underline{\tilde{e}}\rceil \rightsquigarrow \underline{\tilde{e}} : \tau$ is a principal typing for e.

Now let us turn to the properties of Π. Thus assume that $\pi \in \Pi$. We proceed by case analysis on the form of the constraint. We will only consider the case where $\pi \equiv \eta_0 \leq \eta_1$. The other cases follow in the same manner. Now since $\eta_0 \in \ulcorner \eta_0 \leq \eta_1 \urcorner^-$ we know that

$$\eta_0 \in \ulcorner(\Delta, \underline{\tilde{e}}, \tau)\urcorner^- \cup \{0_\xi\} \cup \{\xi \oplus \kappa \mid \xi \in \ulcorner(\Delta, \underline{\tilde{e}}, \tau)\urcorner^-, \kappa \in \ulcorner\underline{\tilde{e}}\urcorner\}$$

and we know that the annotations in $(\Delta, \underline{\tilde{e}}, \tau)$ only consist of annotation variables (since $(\Delta, \underline{\tilde{e}}, \tau) = \texttt{decorate}(\Omega, \underline{e}, \delta)$) so

$$\eta_0 \equiv i$$

for some i. Similarly, since $\eta_1 \in \ulcorner \eta_0 \leq \eta_1 \urcorner^+$ we know that

$$\eta_1 \in \ulcorner(\Delta, \underline{\tilde{e}}, \tau)\urcorner^+ \cup \{0_\eta\} \cup \{\eta \oplus \kappa \mid \eta \in \ulcorner(\Delta, \underline{\tilde{e}}, \tau)\urcorner^+, \kappa \in \ulcorner\underline{\tilde{e}}\urcorner\}.$$

Thus either

$$\eta_1 \equiv i \text{ where } i \in \ulcorner(\Delta, \underline{\tilde{e}}, \tau)\urcorner^+,$$

$$\eta_1 \equiv 0$$

or

$$\eta_1 \equiv i \oplus k \text{ where } i \in \ulcorner(\Delta, \underline{\tilde{e}}, \tau)\urcorner^+ \text{ and } k \in \ulcorner\underline{\tilde{e}}\urcorner.$$

Thus $\eta_0 \leq \eta_1$ is of the form ranged over by ϕ. Now since $\eta_1 \in \ulcorner \eta_0 \leq \eta_1 \urcorner^+$ we are also required to show that if $\eta_1 \equiv i$ or $\eta_1 \equiv i \oplus k$ then $i \notin \ulcorner\underline{\tilde{e}}\urcorner$. We already know that $i \in \ulcorner(\Delta, \underline{\tilde{e}}, \tau)\urcorner^+$ and by the definition of $\ulcorner(\Delta, \underline{\tilde{e}}, \tau)\urcorner^+$ we know that

$$i \in \ulcorner\Delta\urcorner^- \cup \{\xi \mid \xi \in \ulcorner\underline{\tilde{e}}\urcorner\} \cup \{\kappa \mid \kappa \in \ulcorner\underline{\tilde{e}}\urcorner\} \cup \ulcorner\text{types}(\underline{\tilde{e}})\urcorner^+ \cup \ulcorner\text{types}(\underline{\tilde{e}})\urcorner^- \cup \ulcorner\tau\urcorner^+$$

Thus all η's in $\ulcorner(\Delta, \underline{\tilde{e}}, \tau)\urcorner^+$ come from the types and since the annotations in the term are distinct from the annotations in the types (a property of `decorate`) we know that

$$i \notin \ulcorner \underline{\tilde{e}} \urcorner$$

as required.

5.6 Solving the constraints

In this section we will show that the constraints generated by the principal typing algorithm always have an optimal model. By an optimal model we mean a model that generates the best annotated well-typed term according to the ordering specified in section 5.1. We will also give an algorithm that efficiently computes the optimal model and argue it correct.

5.6.1 Existence of optimal models

We start by defining a preorder on closing substitutions indexed by an annotated expression. Let $\tilde{\vartheta}$ and $\tilde{\vartheta}'$ be two closing substitution. We will write $\tilde{\vartheta} \precsim_{\underline{\tilde{e}}} \tilde{\vartheta}'$ iff $\lfloor \underline{\tilde{e}} \tilde{\vartheta} \rfloor \preceq \lfloor \underline{\tilde{e}} \tilde{\vartheta}' \rfloor$. We will say that $\tilde{\vartheta}$ is an optimal model of Φ (which ranges over sets of restricted constraints) with respect to $\precsim_{\underline{\tilde{e}}}$ iff $\tilde{\vartheta} \models \Phi$ and $\tilde{\vartheta} \precsim_{\underline{\tilde{e}}} \tilde{\vartheta}'$ for any $\tilde{\vartheta}'$ such that $\tilde{\vartheta}' \models \Phi$. Note that the preorder ignores the annotation variables that do not occur in $\ulcorner \underline{\tilde{e}} \urcorner$ which means that the variables that occur in the types in $\underline{\tilde{e}}$ are ignored. The reason for this is of course that the annotations in the types do not have any operational significance and we therefore do not care about them as long as the term is well-typed. Although the variables that do not occur in $\ulcorner \underline{\tilde{e}} \urcorner$ are ignored by $\tilde{\vartheta} \precsim_{\underline{\tilde{e}}} \tilde{\vartheta}'$ they may occur in the constraint set. This complicates matters a bit and we have to treat these variables very carefully. Also note that $\precsim_{\underline{\tilde{e}}}$ is covariant in the k's and j's that occur in $\ulcorner \underline{\tilde{e}} \urcorner$ and contravariant in the i's in $\ulcorner \underline{\tilde{e}} \urcorner$. Thus we will try to minimise the k's and the j's and maximise the i's.

We will let Θ range over sets of closing substitutions and we will write $\Theta \models \Phi$ iff all closing substitutions in Θ models Φ. We then define an operator $\widetilde{\sqcap}_{\underline{\tilde{e}}}\Theta$ on these sets.

$$\begin{aligned}
(\widetilde{\sqcap}_{\underline{\tilde{e}}}\Theta)(k) &= \sqcap\{\tilde{\vartheta}(k) \mid \tilde{\vartheta} \in \Theta\} \\
(\widetilde{\sqcap}_{\underline{\tilde{e}}}\Theta)(j) &= \sqcap\{\tilde{\vartheta}(j) \mid \tilde{\vartheta} \in \hat{\Theta}\} \text{ if } j \in \ulcorner \underline{\tilde{e}} \urcorner \\
(\widetilde{\sqcap}_{\underline{\tilde{e}}}\Theta)(j) &= \sqcap\{\tilde{\vartheta}(j) \mid \tilde{\vartheta} \in \Theta\} \text{ if } j \notin \ulcorner \underline{\tilde{e}} \urcorner \\
(\widetilde{\sqcap}_{\underline{\tilde{e}}}\Theta)(i) &= \sqcup\{\tilde{\vartheta}(i) | \tilde{\vartheta} \in \hat{\Theta}\}
\end{aligned}$$

where

$\hat{\Theta} = \{\tilde{\vartheta} \mid \tilde{\vartheta} \in \Theta$ and for any $k \in \ulcorner \underline{\tilde{e}} \urcorner$ it is the case that $\tilde{\vartheta}(k) = \sqcap\{\tilde{\vartheta}(k) \mid \tilde{\vartheta} \in \Theta\}\}$.

Note that the j's that occur in $\ulcorner \underline{\tilde{e}} \urcorner$ are treated differently from the j's that do not. This is best understood in terms of the properties we want to hold for the operator. As the notation suggests we want $\widetilde{\sqcap}_{\underline{\tilde{e}}}$ to be a greatest lower bound operator with respect to $\precsim_{\underline{\tilde{e}}}$. This leaves us with no choice in the definition of $(\widetilde{\sqcap}_{\underline{\tilde{e}}}\Theta)(j)$ when $j \in \ulcorner \underline{\tilde{e}} \urcorner$ but puts no constraint on the definition when $j \notin \ulcorner \underline{\tilde{e}} \urcorner$ (since $\precsim_{\underline{\tilde{e}}}$ ignores the j's such that $j \notin \ulcorner \underline{\tilde{e}} \urcorner$). That $\widetilde{\sqcap}_{\underline{\tilde{e}}}$ really is a greatest lower bound operator is stated in the following lemma.

Lemma 5.6.1
$\widetilde{\sqcap}_{\underline{\tilde{e}}}\Theta$ is a greatest lower bound of Θ with respect to $\precsim_{\underline{\tilde{e}}}$.

Proof 5.6.2
By simple verification.

We also want that, given a set of models Θ, then $\widetilde{\sqcap}_{\underline{\tilde{e}}}\Theta$ should be a model. This is sometimes called the Moore family property [NNH98]. The definition of $(\widetilde{\sqcap}_{\underline{\tilde{e}}}\Theta)(j)$ when $j \notin \ulcorner \underline{\tilde{e}} \urcorner$ is carefully designed to make this hold and is best understood by the proof of the property.

Lemma 5.6.3 (Moore family property)
If $(\Phi, \Delta, \Gamma, \underline{e}, \underline{\tilde{e}}, \tau) = \mathtt{principal}(e)$ and $\Theta \models \Phi$ then $\widetilde{\sqcap}_{\underline{\tilde{e}}}\Theta \models \Phi$.

Proof 5.6.4
This is a nontrivial property that crucially depends on the different treatment of the different sort of j's and on the form of the constraints generated by $\mathtt{principal}(e)$. The proof is by case analysis on the form of the constraints in Φ. Thus assume that $\phi \in \Phi$. We will only consider the two illustrative cases where $\phi \equiv \mathtt{if}\ j > 0\ \mathtt{then}\ k = !$ and $\phi \equiv i_0 \leq i_1 \oplus k$.

case $\phi \equiv \mathtt{if}\, j > 0 \mathtt{then} k = !$**:** We need to show that $\widetilde{\sqcap}_{\underline{\tilde{e}}}\Theta \models \mathtt{if}\, j > 0 \mathtt{then} k = !$. Now since $j \in \ulcorner \mathtt{if}\ j > 0\ \mathtt{then}\ k = !^{\neg -} \urcorner$ we know by the principal typing theorem that $j \notin \ulcorner \underline{\tilde{e}} \urcorner$. Thus, by the definition of $\widetilde{\sqcap}_{\underline{\tilde{e}}}\Theta$, to show that $\widetilde{\sqcap}_{\underline{\tilde{e}}}\Theta \models \mathtt{if}\ j > 0\ \mathtt{then}\ k = !$ amounts to showing that

$$\text{if } \sqcap\{\tilde{\vartheta}(j) \mid \tilde{\vartheta} \in \Theta\} > 0 \text{ then } \sqcap\{\tilde{\vartheta}(k) \mid \tilde{\vartheta} \in \Theta\} = !.$$

Thus assume that

$$\sqcap\{\tilde{\vartheta}(j) \mid \tilde{\vartheta} \in \Theta\} > 0.$$

Then for any $\tilde{\vartheta} \in \Theta$ we know that

$$\tilde{\vartheta}(j) > 0$$

and since $\tilde{\vartheta} \models \mathtt{if}\ j > 0\ \mathtt{then}\ k = !$ we may conclude that

$$\tilde{\vartheta}(k) = !$$

for any $\tilde{\vartheta} \in \Theta$. Thus

$$\sqcap\{\tilde{\vartheta}(k) \mid \tilde{\vartheta} \in \Theta\} = !$$

as required.

case $\phi \equiv i_0 \leq i_1 \oplus k$**:** We need to show that $\widetilde{\sqcap}_{\underline{\tilde{e}}}\Theta \models i_0 \leq i_1 \oplus k$. That is to show that $\sqcup\{\tilde{\vartheta}(i_0) \mid \tilde{\vartheta} \in \hat{\Theta}\} \leq \sqcup\{\tilde{\vartheta}(i_1) \mid \tilde{\vartheta} \in \hat{\Theta}\} \oplus \sqcap\{\tilde{\vartheta}(k) \mid \tilde{\vartheta} \in \Theta\}$. Now for any $\tilde{\vartheta} \in \hat{\Theta}$ we know that

$$\tilde{\vartheta}(i_0) \leq \tilde{\vartheta}(i_1) \oplus \tilde{\vartheta}(k) \leq \sqcup\{\tilde{\vartheta}(i_1) \oplus \tilde{\vartheta}(k) \mid \tilde{\vartheta} \in \hat{\Theta}\}.$$

Thus

$$\sqcup\{\tilde{\vartheta}(i_0) \mid \tilde{\vartheta} \in \hat{\Theta}\} \leq \sqcup\{\tilde{\vartheta}(i_1) \oplus \tilde{\vartheta}(k) \mid \tilde{\vartheta} \in \hat{\Theta}\}.$$

Now since $i_1 \oplus k \in \ulcorner i_0 \leq i_1 \oplus k \urcorner^+$ the principal typing theorem gives us the key fact that

$$k \in \ulcorner \underline{\tilde{e}} \urcorner$$

and thus by the definition of $\hat{\Theta}$ we know that for any $\tilde{\vartheta} \in \hat{\Theta}$

$$\tilde{\vartheta}(k) = \sqcap\{\tilde{\vartheta}(k) \mid \tilde{\vartheta} \in \Theta\}$$

Thus

$$\sqcup\{\tilde{\vartheta}(i_1) \oplus \tilde{\vartheta}(k) \mid \tilde{\vartheta} \in \hat{\Theta}\} = \sqcup\{\tilde{\vartheta}(i_1) \oplus \sqcap\{\tilde{\vartheta}(k) \mid \tilde{\vartheta} \in \Theta\} \mid \tilde{\vartheta} \in \hat{\Theta}\}$$

and

$$\sqcup\{\tilde{\vartheta}(i_1) \oplus \sqcap\{\tilde{\vartheta}(k) \mid \tilde{\vartheta} \in \Theta\} \mid \tilde{\vartheta} \in \hat{\Theta}\} = \sqcup\{\tilde{\vartheta}(i_1) \mid \tilde{\vartheta} \in \hat{\Theta}\} \oplus \sqcap\{\tilde{\vartheta}(k) \mid \tilde{\vartheta} \in \Theta\}$$

So we may conclude that

$$\sqcup\{\tilde{\vartheta}(i_0) \mid \tilde{\vartheta} \in \hat{\Theta}\} \leq \sqcup\{\tilde{\vartheta}(i_1) \mid \tilde{\vartheta} \in \hat{\Theta}\} \oplus \sqcap\{\tilde{\vartheta}(k) \mid \tilde{\vartheta} \in \Theta\}$$

as required.

We can now show that there exists an optimal model for the restricted form of constraints generated by $\texttt{principal}(e)$.

Proposition 5.6.5 (Existence of optimal models)
If $(\Phi, \Delta, \Gamma, \underline{e}, \underline{\tilde{e}}, \tau) = \texttt{principal}(e)$ then there exists an optimal model $\tilde{\vartheta}$ for Φ.

Proof 5.6.6
Assume $(\Phi, \Delta, \Gamma, \underline{e}, \underline{\tilde{e}}, \tau) = \texttt{principal}(e)$ and let $\tilde{\vartheta} = \widetilde{\sqcap}_{\underline{\tilde{e}}}\{\tilde{\vartheta} \mid \tilde{\vartheta} \models \Phi\}$. Then $\tilde{\vartheta}$ is a model of Φ by the Moore family property and $\tilde{\vartheta}$ is smaller that any other model (with respect to $\precsim_{\underline{\tilde{e}}}$) since $\widetilde{\sqcap}_{\underline{\tilde{e}}}$ is a greatest lower bound operator.

5.6.2 Rewriting the constraint set

Now we will go on and show how to efficiently compute the optimal solution to a set of constraints. The algorithm is divided into three phases. The first phase is phrased as a rewrite system which extends the constraint set. When the rewriting has terminated the second phase simplifies the constraints to a simple form which can be solved easily in the third phase.

For the purpose of the algorithm we also include constraints of the form $j > 0$. The idea behind the first phase is to find all k's that is forced to be ! in every model of the constraints. Then if k is forced to be ! in every model we record that by adding an explicit constraints $k = !$. A k can be forced to ! for three reasons: either explicitly by a constraint $k = !$, or implicitly by a constraint $k' \leq k$ where k' is forced to be !, or by a constraint $\mathtt{if}\, j > 0 \,\mathtt{then}\, k = !$ where j is forced to be non-zero. A rewrite relation based on this observation is given below.

$$
\begin{array}{lcll}
\Phi & \longrightarrow & \Phi,\, j' > 0 & \text{if } j \oplus k \leq j',\, k = ! \in \Phi,\, j' > 0 \notin \Phi \\
\Phi & \longrightarrow & \Phi,\, j' > 0 & \text{if } j > 0,\, j \oplus k \leq j' \in \Phi,\, j' > 0 \notin \Phi \\
\Phi & \longrightarrow & \Phi,\, j' > 0 & \text{if } j > 0,\, j \leq j' \in \Phi,\, j' > 0 \notin \Phi \\
\Phi & \longrightarrow & \Phi,\, k = ! & \text{if } j > 0,\, \mathtt{if}\, j > 0 \,\mathtt{then}\, k = ! \in \Phi,\, k = ! \notin \Phi \\
\Phi & \longrightarrow & \Phi,\, k' = ! & \text{if } j > 0,\, \mathtt{if}\, j \oplus k > 0 \,\mathtt{then}\, k' = ! \in \Phi,\, k' = ! \notin \Phi \\
\Phi & \longrightarrow & \Phi,\, k' = ! & \text{if } k = !,\, \mathtt{if}\, j \oplus k > 0 \,\mathtt{then}\, k' = ! \in \Phi,\, k' = ! \notin \Phi \\
\Phi & \longrightarrow & \Phi,\, k' = ! & \text{if } k = !,\, k \leq k' \in \Phi,\, k' = ! \notin \Phi
\end{array}
$$

It is easy to show that the rewrite relation preserves models and that it, given a finite constraint set, terminates with a normal form. This is expressed in the following two lemmas.

Lemma 5.6.7 (Rewriting preserves models)
If $\Phi \longrightarrow \Phi'$ then $\tilde{\vartheta} \models \Phi$ iff $\tilde{\vartheta} \models \Phi'$

Proof 5.6.8
By verifying each rewrite rule.

Lemma 5.6.9
Given a finite set of constraints Φ the rewriting terminates.

Proof 5.6.10
For each rewrite step the number of constraints increases by one. However the number of annotation variables remains constant. Since we only add constraints of the forms $k = !$ and $j > 0$ the number of rewrite steps is bounded by the number of annotation variables.

5.6.3 Simplifying the constraint set

The second phase of the algorithm takes a set of constraints in normal form and simplifies the constraints to a form which can be solved easily. The simplification is based on the idea that if we instantiate a set of constraints Φ with a substitution $\tilde{\theta}$ and find a model $\tilde{\vartheta}$ to the instantiated constraints then we can construct a model $\tilde{\vartheta} \circ \tilde{\theta}$ of the original constraints as well. This is expressed in the following lemma.

Lemma 5.6.11 (Soundness of instantiation)
If $\tilde{\vartheta} \models \Phi\tilde{\theta}$ then $\tilde{\vartheta} \circ \tilde{\theta} \models \Phi$

Proof 5.6.12
By simple case analysis on the form of the constraints in Φ.

It may of course be the case that the instantiated constraints do not have a model. The following lemma however states that if there is a model of the form $\tilde{\vartheta} \circ \tilde{\theta}$ then we could find it with this approach.

Lemma 5.6.13 (Restricted completeness of instantiation)
If $\tilde{\vartheta} \circ \tilde{\theta} \models \Phi$ then $\tilde{\vartheta} \models \Phi\tilde{\theta}$.

Proof 5.6.14
By simple case analysis on the form of the constraints in Φ.

We will now given a constraint set in normal form show how to instantiate the constraint set in a way such that the instantiated constraint set does have models. It will also help us to get rid of all k's in the constraints which will allow us to simplify the constraints considerably. We define $\tilde{\theta}_\Phi$ as follows.

$$\begin{aligned}
\tilde{\theta}_\Phi(i) &= i \\
\tilde{\theta}_\Phi(j) &= 0 \text{ if } j > 0 \notin \Phi \\
\tilde{\theta}_\Phi(j) &= j \text{ if } j > 0 \in \Phi \\
\tilde{\theta}_\Phi(k) &= \checkmark \text{ if } k = ! \notin \Phi \\
\tilde{\theta}_\Phi(k) &= ! \text{ if } k = ! \in \Phi \\
\tilde{\theta}_\Phi(a_i) &= a_i
\end{aligned}$$

We will then go on and solve $\Phi\tilde{\theta}_\Phi$.

The definition of $\tilde{\theta}_\Phi$ is based on the following idea: if for every model $\tilde{\vartheta}$ of Φ it the case that $\tilde{\vartheta}(k) = !$ then this will be explicitly recorded by a constraint $k = !$ in Φ. Thus if $k = ! \notin \Phi$ then we should be able to instantiate k to $\checkmark$ without losing all models. This is not obvious though. If we for example have a constraint $\texttt{if}\ j > 0\ \texttt{then}\ k = !$ in Φ we will by instantiating k to $\checkmark$ force j to be zero which might conflict with another constraint. Fortunately the rewrite

relation ensures that if for every model $\tilde{\vartheta}$ of Φ it is the case that $\tilde{\vartheta}(j) > 0$ then this will be explicitly recorded by a constraint $j > 0$ in Φ and if $j > 0 \in \Phi$ then so would also $k = !$ (remember that we require Φ to be in normal form). Thus $j > 0$ cannot be in Φ so we should also be able to instantiate j to 0. We will shortly show that it is indeed true that $\Phi\tilde{\theta}_\Phi$ do have models and also an optimal model.

This is however not enough. We also need to show that an optimal model of the instantiated constraints generates an optimal model of the original constraints. At first sight it may seem obvious. We are supposed to minimise the k's and the j's and we define $\tilde{\theta}_\Phi(k)$ and $\tilde{\theta}_\Phi(i)$ to be the smallest possible annotations where it is possible and $\tilde{\theta}_\Phi$ is the identity otherwise. However, there may be a constraint $i_0 \leq i_1 \oplus k$ in Φ. Then letting $\tilde{\theta}_\Phi(k) = \checkmark$ may force i_0 to be smaller than otherwise possible and we do want to maximise the i's. This is exactly what happens with the example in section 5.1 which was the motivating example for the decision to first optimise the annotations on the bindings and then optimise the annotations on the value. This is also reflected in the preorder $\precsim_{\underline{\tilde{e}}}$ which allows us to make the annotations on values arbitrary worse if the annotations on the bindings are made better. But $\precsim_{\underline{\tilde{e}}}$ ignores the k's that occur in the types in $\underline{\tilde{e}}$. So if we have a constraint $i_0 \leq i_1 \oplus k$ in Φ where $k \notin \ulcorner \underline{\tilde{e}} \urcorner$, we could lose the optimal model if we let $\tilde{\theta}_\Phi(k) = \checkmark$. Fortunately the principal typing theorem guarantees that if $i_0 \leq i_1 \oplus k \in \Phi$ then $k \in \ulcorner \underline{\tilde{e}} \urcorner$.

Since $\Phi\tilde{\theta}_\Phi$ does not contain any k's we will order the models of $\Phi\tilde{\theta}_\Phi$ by a preorder that ignores the k's. We define the preorder as $\tilde{\vartheta} \lesssim \tilde{\vartheta}'$ iff for all i, j, $\tilde{\vartheta}'(i) \leq \tilde{\vartheta}(i)$ and $\tilde{\vartheta}(j) \leq \tilde{\vartheta}'(j)$.

We are now ready to state and prove that if the instantiated constraint set has an optimal model then so does the original constraint set.

Lemma 5.6.15
If $(\Phi, \Delta, \Gamma, \underline{e}, \underline{\tilde{e}}, \tau) = \texttt{principal}(e)$, $\Phi \longrightarrow^* \Phi'$, Φ' is in normal form and $\tilde{\vartheta}$ is an optimal model of $\Phi'\tilde{\theta}_{\Phi'}$ with respect to $\lesssim$ then $\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'}$ is an optimal model of Φ.

Proof 5.6.16
Assume that $(\Phi, \Delta, \Gamma, \underline{e}, \underline{\tilde{e}}, \tau) = \texttt{principal}(e)$, $\Phi \longrightarrow^* \Phi'$ and that $\tilde{\vartheta}$ is an optimal model of $\Phi'\tilde{\theta}_{\Phi'}$ with respect to $\lesssim$. We first argue that $\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'}$ is a model of Φ as follows. Since $\tilde{\vartheta} \models \Phi'\tilde{\theta}_{\Phi'}$ we know by the soundness of instantiation that

$$\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'} \models \Phi'$$

and since rewriting preserves models we know that

$$\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'} \models \Phi.$$

To show that $\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'}$ is an optimal model, take an arbitrary model $\tilde{\vartheta}'$ of Φ. Then since rewriting preserves models we know that

$$\tilde{\vartheta}' \models \Phi'.$$

We then note that given arbitrary k we know that if $(\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'})(k) = !$ then $k = ! \in \Phi'$ so $\tilde{\vartheta}'(k) = !$ as well (since $\tilde{\vartheta}' \models \Phi'$). Thus we know that

$$(\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'})(k) \leq \tilde{\vartheta}'(k)$$

for all k. Now if there exists a $k \in \ulcorner \underline{\tilde{e}} \urcorner$ such that $(\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'})(k) < \tilde{\vartheta}'(k)$ we are done. Assume therefore that there is no such k, that is

$$(\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'})(k) = \tilde{\vartheta}'(k)$$

for all $k \in \ulcorner \underline{\tilde{e}} \urcorner$. Now let $\tilde{\vartheta}''$ be defined as follows.

$$\begin{aligned}
\tilde{\vartheta}''(k) &= \tilde{\theta}_{\Phi'}(k) \\
\tilde{\vartheta}''(j) &= (\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'}(j)) \sqcap \tilde{\vartheta}'(j) \\
\tilde{\vartheta}''(i) &= (\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'}(i)) \sqcup \tilde{\vartheta}'(i)
\end{aligned}$$

By definition of $\tilde{\theta}_{\Phi'}$ we see that

$$\tilde{\vartheta}' = \tilde{\vartheta}'' \circ \tilde{\theta}_{\Phi'}$$

and from the definition of $\widetilde{\sqcap}_{\underline{\tilde{e}}}$ and $(\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'})(k) = \tilde{\vartheta}'(k)$ for all $k \in \ulcorner \underline{\tilde{e}} \urcorner$ we know that

$$\tilde{\vartheta}'' = (\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'}) \, \widetilde{\sqcap}_{\underline{\tilde{e}}} \, \tilde{\vartheta}'$$

Now, since the set of models is closed under $\widetilde{\sqcap}_{\underline{\tilde{e}}}$

$$(\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'}) \, \widetilde{\sqcap}_{\underline{\tilde{e}}} \, \tilde{\vartheta}' \models \Phi$$

so since rewriting preserves models

$$(\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'}) \, \widetilde{\sqcap}_{\underline{\tilde{e}}} \, \tilde{\vartheta}' \models \Phi'$$

so by $\tilde{\vartheta}'' = (\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'}) \, \widetilde{\sqcap}_{\underline{\tilde{e}}} \, \tilde{\vartheta}'$

$$\tilde{\vartheta}'' \models \Phi'$$

and by $\tilde{\vartheta}'' = \tilde{\vartheta}'' \circ \tilde{\theta}_{\Phi'}$

$$\tilde{\vartheta}'' \circ \tilde{\theta}_{\Phi'} \models \Phi'$$

and therefore

$$\tilde{\vartheta}'' \models \Phi' \tilde{\theta}_{\Phi'}$$

by the partial completeness of instantiation. But $\tilde{\vartheta}$ is an optimal model of Φ' with respect to $\lesssim$ so

$$\tilde{\vartheta} \lesssim \tilde{\vartheta}''.$$

Thus for any i,

$$\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'}(i) = \tilde{\vartheta}(i) \geq \tilde{\vartheta}''(i) = (\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'}(i)) \sqcup \tilde{\vartheta}'(i) \geq \tilde{\vartheta}'(i)$$

and for any j

$$\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'}(j) \leq \tilde{\vartheta}(j) \leq \tilde{\vartheta}''(j) = (\tilde{\vartheta} \circ \tilde{\theta}_{\Phi'}(j)) \sqcap \tilde{\vartheta}'(j) \leq \tilde{\vartheta}'(j)$$

as required.

Now let us return to the task of showing that the instantiated constraint set has an optimal model. We will do that by showing that the we can construct sets of constraints of a simple form which are equivalent to the instantiated constraint set. We will then argue that for these constraint sets it is straightforward to find an optimal model.

Let ψ, υ and ς range over constraints of the following forms.

$$\begin{array}{rcl} \psi & ::= & i \leq 0 \mid i_0 \leq i_1 \mid i_0 \leq i_1 + 1 \\ \upsilon & ::= & j_0 \leq j_1 \mid j_0 + 1 \leq j_1 \mid j > 0 \\ \varsigma & ::= & a_i \leq a_j \end{array}$$

We will let Ψ, Υ, Σ range over set of constraint of the form ranged over by ψ, υ and ς respectively. Now the following result states that the instantiated constraint set can be simplified.

Lemma 5.6.17 (Simplification)
If Φ is a finite set of constraints in normal form then there exist finite sets Ψ, Υ, Σ such that $\tilde{\vartheta} \models \Phi\tilde{\theta}_\Phi$ iff $\tilde{\vartheta} \models \Psi$, $\tilde{\vartheta} \models \Upsilon$ and $\tilde{\vartheta} \models \Sigma$.

Proof 5.6.18
Assume Φ is in normal form. We proceed by proving that for any constraint ϕ in Φ either $\phi\tilde{\theta}_\Phi$ is trivially true (ie it is modelled by every closing substitution) or there exists an equivalent constraint of the form given above. Thus given a constraint ϕ we proceed by case analysis on the form of the constraint. We will only consider the illustrative case where $\phi \equiv j_0 \oplus k \leq j_1$. Let us start by considering the case where $\tilde{\theta}_\Phi(k) = \checkmark$. Then if $\tilde{\theta}_\Phi(j_0) = 0$ we have that

$$(j_0 \oplus k \leq j_1)\tilde{\theta}_\Phi \equiv 0 \oplus \checkmark \leq \tilde{\theta}_\Phi(j_1)$$

which is trivially satisfiable. Consider instead the case where $\tilde{\theta}_\Phi(j_0) = j_0$. By the definition of $\tilde{\theta}_\Phi$,

$$j_0 > 0 \in \Phi$$

so

$$j_1 > 0 \in \Phi$$

since Φ is in normal form. Thus by the definition of $\tilde{\theta}_\Phi$ we know that

$$\tilde{\theta}_\Phi(j_1) = j_1.$$

Thus

$$(j_0 \oplus k \leq j_1)\tilde{\theta}_\Phi \equiv j_0 \oplus \checkmark \leq j_1 \equiv j_0 \leq j_1$$

which is among the constraints of the form above. Now consider the case where $\tilde{\theta}_\Phi(k) = !$. Then by the definition of $\tilde{\theta}_\Phi$,

$$\kappa = ! \in \Phi$$

so

$$j_1 > 0 \in \Phi$$

since Φ is in normal form. Thus by the definition of $\tilde{\theta}_\Phi$ we know that

$$\tilde{\theta}_\Phi(j_1) = j_1.$$

Now in the case where $\tilde{\theta}_\Phi(j_0) = 0$ we have that

$$(j_0 \oplus k \leq j_1)\tilde{\theta}_\Phi \equiv 0 \oplus ! \leq j_1$$

which is equivalent to $j_1 > 0$. In the case where $\tilde{\theta}_\Phi(j_0) = j_0$ we have that

$$(j_0 \oplus k \leq j_1)\tilde{\theta}_\Phi \equiv j_0 \oplus ! \leq j_1$$

which is equivalent to $j_0 + 1 \leq j_1$.

5.6.4 Solving the simplified constraints

It remains to show that there are algorithms that can find optimal models of simplified forms of constraint sets. These algorithms are straightforward and we will only describe them informally. It should be straightforward to describe them as rewrite systems as well.

To solve constraints ranged over by ψ we proceed as follows. First remember that ψ ranges over constraints of the form

$$\psi ::= \quad i \leq 0 \mid i_0 \leq i_1 \mid i_0 \leq i_1 + 1$$

and that we are supposed to maximise the i's. The first stage of the algorithm will find all i's that has to be 0. Either directly due to a constraint $i \leq 0$ or indirectly due to a constraint $i_0 \leq i_1$ where i_1 is forced to be 0. We can express this as a rewrite system with the single rewrite rule

$$\Psi \longrightarrow \Psi, i' \leq 0 \quad \text{if } i \leq 0, i' \leq i \in \Psi,\ i' \leq 0 \notin \Psi.$$

We apply this rule until we reach a normal form. Then we instantiate the constraints by substituting in 0 for i if $i \leq 0 \in \Psi$. We can then simplify the instantiated constraints to constraints of the following form.

$$i \leq 1 \mid i_0 \leq i_1 \mid i_0 \leq i_1 + 1$$

We see that they have the same form as before except that $i \leq 0$ is replaced by $i \leq 1$. Thus in the second stage we will try to find all i's such that $i \leq 1$

and instantiate them to 1. We repeat this process until we have a constraint set where there is no constraint of the form $i \leq n$. We can then take the remaining i's to be ω. The algorithm will terminate since the number of variables in the constraint set is strictly decreasing for each stage. Moreover the algorithm will be linear in the number of i's in Ψ since the cost of each stage is proportional to the number of eliminated variables.

To solve constraints ranged over by υ we proceed as follows. First remember that υ ranges over constraints of the form

$$\upsilon ::= \quad j_0 \leq j_1 \mid j_0 + 1 \leq j_1 \mid j > 0$$

and that we are supposed to minimise the j's. Also note that any variable j that occurs in the constraint set is constrained by a constraint $j > 0$ (this follows from the definition of $\tilde{\theta}_\Phi$) so the best value we can assign to a j is 1. Now think of the constraint set as a graph where the j's are nodes and the constraints are edges. First find all the cycles in the graph that only contain edges of the form $j_0 \leq j_1$. The variables in such a cycles must be equal so we can instantiate them all to a common variable. When there is no cycle of this form left we find all remaining cycles. That is cycles that contain an edge of the form $j_0 + 1 \leq j_1$. All the variables in such a cycle must be ω so we instantiate them to ω. Some constraints can then be simplified away and we will have a constraint set of the form

$$\omega \leq j_1 \mid j_0 \leq j_1 \mid j_0 + 1 \leq j_1 \mid j > 0$$

that do not contain any cycles. We then find all j's that has to be ω, instantiate, simplify and remove trivial constraints. We will then again have constraints of the form

$$j_0 \leq j_1 \mid j_0 + 1 \leq j_1 \mid j > 0$$

without cycles. Since there are no cycles in the graph there must be some variables that do not occur in the right hand side of any constraint. Instantiate them all to 1 and simplify. We will get a constraint set of the form

$$j_0 \leq j_1 \mid j_0 + 1 \leq j_1 \mid j > 1.$$

where every variable j in the constraint set is constrained by a constraint $j > 1$. We then repeat this process until the set of constraints is empty. This will eventually happen since the number of variables in the constraint set is strictly decreasing for each of the repeated stages. Moreover the algorithm will be linear in the number of j's in Υ since the cost of each stage is proportional to the number of eliminated variables.

Finally, we can solve the constraints ranged over by ς by instantiating the involved bare type variables to `Int`.

5.7 A note on complexity

The complexity of the implementation of the analysis depends on the complexity of the three phases of the algorithm. Most compilers perform type inference anyway so we will assume that this phase has been carried out and that the input of the analysis is an explicitly typed term.

At first sight it might seem as if the number of generated constraints is linear in the size of the explicitly typed term but it is not. This claim was made made for the similar type system by Wansbrough and Peyton Jones [WJ99b] but turned out to be false [1]. The non-linearity comes from the rule

$$\text{Value}\frac{\Pi\ ;\ \Gamma \vdash \underline{v} \rightsquigarrow \tilde{\underline{v}} : \rho \qquad \Pi \vdash \texttt{if}\ \xi > 0\ \texttt{then}\ \Gamma \to^* \Gamma, \Gamma}{\Pi\ ;\ \Gamma \vdash \underline{v} \rightsquigarrow \tilde{\underline{v}}^{\iota} : \rho^{[\eta,\xi]} \qquad \Pi \vdash [\eta, \xi] \leq \iota}$$

where the side condition $\Pi \vdash \texttt{if}\ \xi > 0\ \texttt{then}\ \Gamma \to^* \Gamma, \Gamma$ generates one constraint for every free variable in Γ. Thus the number of constraints generated for

$$\lambda x_0.\lambda x_1.\ldots.\lambda x_n.x_0 + x_1 + \ldots + x_n$$

is quadratic in n. Thus the overall complexity of the analysis is at least quadratic in the size of the explicitly typed program (and the same holds for the type systems by Turner et al [TWM95] and Wansbrough and Peyton Jones [WJ99b]). However in practice the number of nested abstractions does not grow with program size. Under that assumption we believe that it might be the case that the number of constraints is linear in the size of the explicitly typed term. We leave the investigation of this as future work.

It remains to show that we can find the optimal solution to the constraints in linear time. We have proved that given a constraint set of size n the rewrite system that underly our implementation terminates in n steps. However we have not argued that the rewrite system can be implemented in linear time although we believe it can.

[1] Confirmed by personal communication.

Chapter 6

Experiments

We have made a prototype implementation of the analysis and an interpreter of the abstract machine that counts the number of updates and update marker checks. The implementation follows the semantics of chapter 2 with two exceptions. First we have implemented the rule

$$\langle H, x = v\ ;\ x\ ;\ S\rangle \overset{\text{Lookup}}{\longmapsto} \langle H, x = v\ ;\ v\ ;\ S\rangle$$

which allows a value to be looked up without pushing a marker and performing an update. Any reasonable compiler would perform this optimisation so we have included it in our implementation to make for more realistic benchmarks. Second, we have implemented the known function call optimisation which reduces the number of update marker checks when calling top-level functions. The known function call optimisation works as follows. Suppose, for example, that we have a top-level function f of arity 3 which is called as $f\,x\,y\,z$. The first thing the code for f will do is to look for update markers on the top of the stack. But it will not find any since there will be three arguments on the top of the stack. At the call of f we know that and can take advantage of it by simply jumping to the code right after the code that performs the update marker checks. Similarly if f is called with just two arguments (as in $\texttt{let}\ g = f\ x\ y\ \texttt{in}\ \mathit{map}\ g\ \mathit{xs}$) we can bypass parts of the code that checks for markers. The optimisation only applies to top-level functions since it requires that we statically know where the code for f resides. To increase the possibility for the optimisation we have implemented a simple transformation that floats functions to the top level where it is possible (that is when they do not have any free variables except for those referring to other top level declarations). This transformation can also save a few updates since it can turn

$$\begin{array}{l}\texttt{let}\ f = \texttt{let}\ g = \lambda y.e_1\\ \qquad\qquad\ \ \texttt{in}\ \lambda x.e_0\\ \texttt{in}\ e\end{array}$$

into

$$\begin{array}{l}\texttt{let}\ f = \lambda x.e_0 \\ \quad\ \ g = \lambda y.e_1 \\ \texttt{in}\ e\end{array}$$

which makes f bound to a value. This explains why the measurements presented here do not exactly coincide with those in [Gus98]. The known function call optimisation seems to go back to at least the implementation of the three instruction machine which as far as we know is the first abstract machines based on update markers [FW87].

We have measured the effects of our analysis for a few small Haskell programs. The program sizes range from two lines up to thirty lines (the most substantial ones being a byte code interpreter and a calculator that parses arithmetic expressions). Some of the programs heavily use Haskell's powerful syntax (such as list comprehensions) and several predefined functions and when translated into our restricted language the program sizes range from 12 lines up to 190 lines (including the used predefined functions). In appendix C we show one of the programs and the result of applying the analysis to it. The constraint solver in our implementation (written in Haskell) is very naive and is at least quadratic in the number of constraints. Anyhow the largest program can be analysed within three seconds. We believe that with a good implementation of the constraint solver (which we believe can be linear in the number of constraints) large programs can be analysed quickly.

The results of the measurements are given below.

	Without analysis		With analysis		Saved %	
Program	Upd	Chk	Upd	Chk	Upd	Chk
primes	6685	48983(61416)	639	0(0)	90	100(100)
substring	102	370(502)	63	1(12)	38	100(98)
nqueens	712	2446(3760)	72	0(0)	90	100(100)
quicksort	38	220(438)	0	0(0)	100	100(100)
interpreter	3254	20909(27088)	3249	431(431)	0	98(98)
zantema	71	664(1005)	66	0(0)	7	100(100)
syracuse	166	1224(1475)	83	0(0)	50	100(100)
calculator	19	315(413)	0	0(0)	100	100(100)

The first and second column contains the number of updates and update marker checks needed in the unoptimised and optimised abstract machine respectively. The third column shows the percentage of updates and update marker checks saved. The figures within parentheses shows the number of update marker checks with the known function call optimisation turned off.

The number of saved updates varies greatly from program to program depending on the amount of inherent sharing and ranges from 0% to 100% with an average of 59%. Thus for a fair comparison with other analyses it is crucial to compare the results for the same programs. The measurements in [Ses91]

are performed on very small programs and furthermore functions are used to encode lists. He also uses an abstract machine which treats updates differently (see chapter 7 for a discussion of this). His results range from 0% to 53% with an average of 25%. The measurements in [Mar93] are performed on large real world programs which we cannot match. His results range from 0% to 57% with an average of 23%. The measurements by Faxén [Fax96] are carried out on a few small programs. The results vary from 0% to 100% with an average of 66%. Although a direct comparison is not possible, our results seem promising.

The number of saved update marker checks is constantly over 98% with an average of 99% and seems to be independent of the nature of the program. The measurements by [Ses91] show that the number of update marker checks that can be saved by his analysis ranges from 0% to 55% with an average of 25%. We also note that the known function call optimisation can reduce the number of checks with between 17% and 50% with an average of 29%. Together with our analysis the known function call optimisation gives little extra but it is not completely subsumed by our analysis.

These initial results seem very promising but indeed the programs tried out are very small. Most notably, none of them suffer from the lack of polymorphism and polyvariance (see chapter 8 for a discussion of polymorphism and polyvariance). It would be very interesting to incorporate the analysis in a real compiler to see how it behaves on large programs and to measure the actual speedup.

Chapter 7

Related work

Our type systems builds on ideas taken from linear logic [Gir87] and is yet another attempt to exploit the computational interpretations of linear logic [Abr93]. There are a number of type systems and analyses building on linear logic and where there is a connection to our work we will relate to them later in this chapter. The chapter is organised as follows. First we will relate our approach to those analyses which also tackle the problem of update avoidance and the problem of avoiding update marker checks. Then we will relate to work which tackles the closely related problems of ensuring the safety of program transformation, destructively updating arrays and compile time garbage collection. Finally, we will relate sharing analysis to strictness analysis.

7.1 Avoiding updates

The idea of avoiding unnecessary updates is old and goes back at least to Fairbairn [Fai85] but he gives no analysis. Also as pointed out by Fairbairn and Wray [FW87], and Burn, Peyton Jones and Robson [BRJ88] the abstract machines that were underlying the implementations in those days were not very well suited for exploiting sharing analysis. One of the main objectives behind the design of the Three Instruction Machine(TIM) by Fairbairn and Wray [FW87] and the Spineless G-machine by Burn, Peyton Jones and Robson [BRJ88] was therefore to open up the possibility for sharing analysis. Fairbairn and Wray are also the first to report on a simple local sharing analysis which can speed up the TIM by about 10%. However the TIM has a fairly naive treatment of sharing which can lead to long chains of indirections (corresponding to bindings of the form $x = y$). When such a chain is evaluated all the bindings are updated. The good results from Fairbairns and Wray's analysis seem to stem mainly from the elimination of some of these chains. The TIM was later refined into the G-TIM by Argo [Arg89]. In the G-TIM these chains are not created and Fairbairn and Wray's analysis does not seem applicable.

As far as we know the first non local analyses for update avoidance are a backwards analysis by Hughes and Wray [Hug88] and a path analysis by Bloss and Hudak [BHY88].

Hughes and Wray's analysis is based on counting the number of times an expression is used under call-by-name which approximates its use under call-by-need. If the analysis can figure out that an expression is used at most once then it is safe not to update the corresponding closure. Since their analysis is based on call-by-name rather than call-by-need it is rather conservative. Hughes and Wray give their analysis for a first order language without data structures. However it seems possible to extend it to a higher order language with data structures although such an analysis would be very expensive.

Bloss and Hudak's path analysis can predict the order of evaluation of variables in an expression. Consider for example

$$\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ y + y$$

and rename the different occurrences of y as below.

$$\texttt{if}\ x\ \texttt{then}\ y_0\ \texttt{else}\ y_1 + y_2$$

If we apply path analysis to this expression we would find out that either x is needed first and then y_0 or x is needed first and then y_1 and y_2. Based on this information we could annotate the expression as

$$\texttt{if}\ x^{\checkmark}\ \texttt{then}\ y^{\checkmark}\ \texttt{else}\ y + y$$

where the $\checkmark$ indicates that the binding referred to by x need not be updated and the binding referred to by y need not be updated if the `then`-branch is selected. Note that this annotation schema is quite different from that in our analysis where we annotate bindings rather than variables. Our choice to annotate bindings rather than variables reflects our intention to base our implementation of the abstract machine on the so called self-updating model (used in for example the STG-machine [PJ92]) rather than the so called cell-model where the responsibility to update the closure lies on the code that forces the evaluation of the closure (ie the code for the variable). An analysis for the self-updating model (like ours) can in this example not annotate the binding for y with $\checkmark$ since it would go wrong if the `else`-branch is taken. Thus analyses based on the cell-model can avoid more updates in this case. However for some examples it can also be the other way around. Unfortunately the path analysis by Bloss and Hudak cannot handle data structures such as lists and it is computationally very expensive. Gomard's and Sestoft's evaluation order analysis [GS91] is a refinement of the work by Bloss and Hudak. Gomard's and Sestoft's analysis can take care of data structures and it is significantly less expensive than Bloss and Hudak's analysis. Unfortunately it is not higher order and seems to be far more expensive than update avoidance analyses not based on evaluation order information.

In his PhD thesis Sestoft presents a so called usage interval analysis [Ses91] which can give a lower and an upper bound on the number of times an expression is used under call-by-name. The lower bound gives strictness information and the upper bound gives sharing information: if we know that an expression is used at most once then we can avoid to update the corresponding closure. Sestoft's analysis is essentially first order but is extended to a higher order language by means of a flow analysis which Sestoft calls closure analysis. In this way he obtains an analysis for a higher order language with a reasonable complexity. The analysis does not treat data structures directly, but they can be encoded as functions without an explosion in time complexity. But due to the encoding the resulting analysis is rather imprecise when it comes to data structures. Although the analysis is quite crude it gives good results and on average 23% of all updates can be avoided. However, Sestoft's measurements is for an abstract machine which accomplishes sharing of evaluation in a rather naive fashion similar to the one in the TIM. It is not clear to us if the good results would carry over to an implementation based on a more efficient abstract machine.

The first type based sharing analysis is due to Launchbury, Gill, Hughes, Marlow, Peyton Jones and Wadler [LGH+92]. It treats non-atomic types (ie function types) conservatively and it cannot handle data structures. Anyhow it has been very influential since it incorporates ideas from linear logic (as proposed by Abramsky [Abr90, Abr93]) and it is the basis for several other type based sharing analyses, including ours.

Marlow presents an analysis based on abstract interpretation [Mar93]. One of the main objectives behind Marlow's work was to construct an analysis which could successfully be implemented in a full scale compiler. Where there was a choice between an efficient analysis and a more accurate one he opted for the former. The result was a rather cheap but fairly conservative analysis (especially when it comes to data structures) which was successfully implemented in the Glasgow Haskell Compiler [JHH+93]. Marlow's measurements show that he can avoid on average about 25% off all unnecessary updates which gives a speedup of about 5%.

Our work builds directly on the type based analysis by Turner, Wadler and Mossin [TWM95]. Their analysis handles a monomorphic language with higher order functions and data structures. The complexity of their analysis appears to be essentially linear in the size of the explicitly typed term (see section 5.7 for a precise statement). Despite the low complexity their analysis is significantly more precise than previous analyses, especially when it comes to data structures. They also prove their analysis sound with respect to Launchbury's natural semantics [Lau93]. Although our work builds closely on the work by Turner et al there are a number of important differences. First, our analysis provides information that enables us to optimise the bookkeeping of updates by avoiding update marker checks. We think of this as our major contribution. Second, their analysis treats only a very restricted form of recursive `let`-expressions. Most notably, they cannot handle mutual recursion. Third, our analysis is strictly

more precise (ie the set of well typed terms in our type system is a strict superset of the well typed terms in their system). One reason for this being that we distinguish between the type of a binding and the type of the corresponding expression. For example,

$$\begin{array}{l}\texttt{let}\ x =^{\checkmark} 1 + 2\ \texttt{in}\\ \texttt{let}\ y =^{!} (\lambda z.z)\ x\ \texttt{in}\\ y + y\end{array}$$

is considered to be ill-typed by their type system, since y has the same type as $(\lambda z.z)\ x$ and thus the same type as x. Another reason for why our type system is more accurate is that our type system has a subsumption rule which allows more terms to be typed. For example,

$$\begin{array}{l}\texttt{let}\ f =^{!} \lambda x.x + 1\\ \quad\ \ y =^{!} 2 + 3\\ \quad\ \ z =^{\checkmark} 4 + 5\\ \texttt{in}\ y + (f\ y) + (f\ z)\end{array}$$

is ill-type in their type system (since it would require f to have two different types). Although our type system is strictly more precise than the one by Turner et al it is not clear to us how big the difference is in practice. However both our refinements (binding types and subsumption) are important for our analysis when it comes to avoiding update marker checks.

Faxén formulates an elegant sharing analysis based on flow information which can be obtained by means of a flow analysis [Fax95]. The formulation of the sharing analysis itself is independent of the flow analysis but the accuracy and complexity of the analysis depends crucially on the accuracy and complexity of the underlying flow analysis. Together with his sharing analysis Faxén presents a suitable flow analysis which can handle a polymorhic higher order language with data structures. The complexity of his flow analysis is not clear to us. Faxén proves his flow analysis correct but gives no proof for the soundness of his sharing analysis. Although Faxén's analysis looks completely different (at least at first sight) from the analysis by Turner et al they are in fact closely related and based on similar ideas. Also the constraint set generated by the type inference algorithm by Turner et al bears great resemblance to the graph that Faxén creates based on the flow information. The exact relationship between the analyses are not clear to us (since the accuracy of Faxén's flow analysis is not clear to us) but it seems to us as if Faxén's analysis yields more precise results in some cases. The examples we have found are exactly those where our analysis is more precise than Turner et al's indicating that our analysis has similar precision to Faxén's when it comes to update avoidance. Faxén makes no attempt to avoid update marker checks. However it should be noted that Faxén's analysis handles a polymorhic language which we do not.

A key feature of Turner et al's, Faxén's and our analysis is that they consider

expressions like

$$\texttt{let}\ x =^{\checkmark} 1 + 2\ \texttt{in}\ x + (\lambda y.3)\ x$$

to be ill-typed even though x is only accessed once. Indeed, taking it to be well-typed would render our analysis unsound since executing it leads to a dangling pointer and thus we consider it to be ill-annotated. However the dangling pointer is not dereferenced during the evaluation and therefore it would make sense to consider it to be well-annotated. To consider programs as ill-annotated only if they actually dereference a dangling pointer is the approach taken by for example Sestoft, Lanchbury et al and Marlow (although they do not state it explicitly). Mogensen takes the type system by Turner et al as his starting point and adopts it to fit this weaker correctness criteria by means of a notion of zero usage [Mog97]. We have chosen to consider programs that leads to dangling pointers as ill-annotated for the following reason: even though a dangling pointer is not dereferenced during the evaluation of the expression, it could very well be dereferenced by the garbage collector. Thus if the analysis cannot guarantee that evaluation does not lead to dangling pointers the implementation must see to that it is safe for the garbage collector to dereference such a pointer. This can be achieved as follows. When a closure that is not to be updated is fetched from the heap the closure is overwritten with a special "dangling pointer closure" that the garbage collector can recognise. If this is not done and the closure is left as it is the garbage collector would retain the closure and all the closures it references which can lead to a space leak. This is very similar to how conventional implementations handle closures that needs to be updated: when they are fetched they are overwritten with a so called black hole closure which the garbage collector can recognise. (Some implementation instead takes care of the problem when the garbage collector is invoked: it then scans the stack to find all update markers and overwrites the corresponding closures with a black hole closure. However this optimisation does not carry over to the "dangling pointer closure" problem, since there are no markers for these closures on the stack.) Although it is easy for an implementation to handle these dangling pointers there is a rather high associated cost. We believe that this cost exceeds the gain that could be made by avoiding more updates. Further experiments may be needed to decide upon this tradeoff.

An earlier version of this type system has been previously presented in [Gus98]. Apart from some minor cosmetic changes (by replacing definitions by equivalent ones) we have made one important modification to the type system. This concerns the rule

$$\text{Value}\frac{\Gamma \vdash v \rightsquigarrow \tilde{v} : \rho \qquad \text{if } \xi > 0 \text{ then } \Gamma \rightarrow^* \Gamma, \Gamma}{\Gamma \vdash v \rightsquigarrow \tilde{v}^{\iota} : \rho^{[\eta,\xi]} \qquad [\eta, \xi] \leq \iota}$$

and the side condition if $\xi > 0$ then $\Gamma \rightarrow^* \Gamma, \Gamma$. Recall that the purpose of the side condition is to ensure that if the value is required to take care of any update markers then it must be allowed to safely duplicate the free variables of the value. In the type system presented in [Gus98] this mechanism was spread

out over several rules. However if we reformulated the rules in the style of this presentation we would end up with the following rule.

$$\text{Value}\frac{\Gamma \vdash v \rightsquigarrow \tilde{v} : \rho}{\Gamma \vdash v \rightsquigarrow \tilde{v}^{[\eta',\xi']} : \rho^{[\eta,\xi]}} \quad \begin{array}{l} \text{if } \xi' > 0 \text{ then } \Gamma \to^* \Gamma, \Gamma \\ [\eta, \xi] \leq [\eta', \xi'] \end{array}$$

Note that the side condition now is if $\xi' > 0$ then $\Gamma \to^* \Gamma, \Gamma$ rather than if $\xi > 0$ then $\Gamma \to^* \Gamma, \Gamma$. Thus in our previous type system the side condition ensured that if the value could take care of an update marker then it must be safe to duplicate the free variables of the value. However, in our current type system the side condition ensures that if the value is required to take care of an update marker then it must be safe to duplicate the free variables. Since the other side condition ensures that $\xi \leq \xi'$ (ie that the value can do what is required) this means that in the current type system more terms are well-typed. However this does not influence the accuracy of the type system since the optimally annotated well-typed term in the current type system is also well-typed in the previous one (since then $\xi = \xi'$). The modification is important for two reasons. First the Moore family property (lemma 5.6.3) does not hold for our previous type system and our proof of the existence of optimal models fails (proposition 5.6.5). (Although the result itself holds. The proof would just be less elegant.) Second, we believe it could be important in a setting with separate compilation. Then when annotating a function that is called from other modules we might want be overly conservative and annotate some values with $[0, \omega]$ so that the function can be called in any (at that point unknown) context. With the previous type system this could lead to unnecessarily constraints on the arguments to the function.

Wansbrough and Peyton Jones also take Turner et al's type system as their starting point [WJ99b]. They make two important changes to Turner et al's system which influence the applicability of their type system. First they extend it to handle an underlying polymorphic language (but they have no notion of annotation polymorphism). Second they extend the analysis to user defined data types. They also make two changes to Turner et al's system which influence the accuracy of their type system. First, they introduce a subtyping relation identical to ours (although they did it independently of [Gus98]). Since they do not have any notion of annotation polymorphism, they believe that subtyping has a major impact on the accuracy of their analysis. Experiments that may confirm this claim are forthcoming [WJ99a]. Note that this is in contrast to our expectations of the subtyping in our type system but we have a system with annotation polymorphism in mind. Second, their type language for data types is less expressive than Turner et al's and they cannot, for example, express that the elements of a list need to be updated but the spine of the list needs not. Thus if one single closure in a huge data structure needs to be updated this propagates to the entire data structure. However they have good reasons for using a restrictive type language. We will discuss that further in chapter 8 when we consider the extension of our type system to handle user defined data types. We are also

working on extending our type system to cope with polymorphism (including annotation polymorphism). We believe that the resulting type system will be strictly more accurate then the type system by Wansbrough and Peyton Jones (the restriction of their type system to the language we can handle certainly is less accurate then our current type system). However it is not clear to us how much this will give in practice. Wansbrough and Peyton Jones prove their analysis sound with respect to a single step version of Launchbury's natural semantics [Lau93].

7.2 Avoiding update marker checks

We are only aware of one analysis that can be used to optimise the bookkeeping of updates, namely the usage interval analysis by Sestoft [Ses91]. His analysis annotates values with (annotations corresponding to) $[0,0]$ and $[0,\omega]$, thus his analysis provides less accurate information than ours. Most notably, the common case $[1,1]$ degenerates to $[0,\omega]$. Indeed, a direct comparison shows that this has a significant impact in practice (see chapter 6).

7.3 Program transformation

A problem very closely related to update avoidance is the problem of ensuring the safety of some program transformations. For example β-reduction

$$(\lambda x.e)\ e' \Rightarrow e[x{:=}e']$$

is not safe in general since it may duplicate work if x occurs several times in e. However if we can guarantee that the function uses its argument at most once then we can safely inline e'. Another useful transformation is the following.

$$\texttt{let}\ x = e\ \texttt{in}\ \lambda y.e' \Rightarrow \lambda y.\texttt{let}\ x = e\ \texttt{in}\ e'$$

This transformation also risks duplicating computation since if the abstraction is used several times e may be computed several times. But if we can guarantee that the function is used at most once then we can safely perform the transformation. To devise an analysis that ensures the safety of these transformations is clearly closely related to the problem of update avoidance and the early work by Goldberg [Gol87] which is used essentially to guarantee the safety of the second of these transformations is very closely related to update avoidance analyses. Also the update avoidance analyses by Turner et al [TWM95] and Wansbrough and Peyton Jones [WJ99b] have both been proposed as solutions to the problem. Given the close relationship between our analysis and these analyses we believe that our analysis could also be used for this purpose. However we have no proof of this and as far as we know proving Goldberg's, Turner et al's and Wansbrough and Peyton Jones analyses sound for this purpose is also still an

open problem. It is an interesting question whether all sound update avoidance analyses can also guarantee the safety of these program transformations and vice versa. That is whether the two problems are actually equivalent. We are currently considering if it is possible to use the techniques in the recent work by Moran and Sands [MS99] in order to prove this.

7.4 Destructive array update and compile time garbage collection

Two problems closely related to update avoidance is the destructive array update problem and the compile time garbage collection problem. The destructive array update problem is about efficiently implementing purely functional arrays. Updating a purely functional array in general forces the entire array to be copied before the update can take place since there may be other references to the array. Destructively updating the array can then lead to a loss of referential transparency. However if there is no other reference to the array it may be updated destructively. The compile time garbage collection problem is about reusing memory without the garbage collector being involved. This can be done if the compiler can detect the last use of a closure in the heap. In for example

$$\begin{array}{l}\texttt{case}\ x\ \texttt{of}\\ \quad \texttt{nil} \Rightarrow \ldots\\ \quad \texttt{cons}\ y\ ys \Rightarrow \ldots\end{array}$$

we can reuse the memory where x is stored if we have the only reference to x. These two problems are closely related to each other and some work attack them both simultaneously. Although closely related to the update avoidance problem there is an important difference. The update avoidance problem requires that a closure that is not to be update is used at most once and we ensure that by enforcing that the reference to the closure is passed around in a single threaded fashion. However an array that shall be updated destructively may very well be used several times but when the array gets updated there may only be one reference to it. The literature on these two problems is substantial and we will make no attempt to cover it all. We just note that judging by the similarity of the problems it seems likely that every piece of work on any of these problems is related to update avoidance in some sense. Most apparent are the connections to type systems based on ideas from linear logic [Gir87]. These can be divided into two kinds. First those which form the basis of a linear programming language. This includes the work by Girard and Lafont [GL87] Lafont [Laf88], Homström [Hol88], Wadler [Wad90], Abramsky [Abr90, Abr93], Wakeling and Runciman [WR91] and Mackie [Mac94]. Second those type systems which form the basis of an automatic compile time program analysis. This includes the work by Guzmán and Hudak [GH90], Wadler [Wad91], Barendsen and Smetsers [BS96], Turner et al [TWM95] and Kobayashi [Kob99]. Strikingly similar to our type system is

the uniqueness type system by Barendsen and Smetsers [BS96]; just take ✓ to be unique and ! to be non-unique. However, there is a very important difference: In our type system a function of type $\tau_{\checkmark} \to \tau'$ will allow its parameter to be non-updating and a function of type $\tau_{!} \to \tau'$ will require its parameter to be updating. In contrast, a function with uniqueness type $\tau_{\text{unique}} \to \tau'$ will require its parameter to be unique and a function of type $\tau_{\text{non-unique}} \to \tau'$ will allow its parameter to be non-unique. This also shows up in the subtyping relation where we have $\tau_{!} \leq \tau_{\checkmark}$ in contrast to $\tau_{\text{unique}} \leq \tau_{\text{non-unique}}$. Thus their type system is unsound with respect to update-avoidance and our type system is unsound with respect to update-in-place.

7.5 Strictness analysis

Strictness analysis [Myc82], as sharing analysis, aims at reducing the overhead of call-by-need. It does so by determining when it is safe to pass arguments to functions evaluated rather than unevaluated. This is safe if we know that the function will use its argument. Thus strictness analysis can be thought of as turning call-by-need into call-by-value while avoiding updates can be considered as turning call-by-need into call-by-name.

Since both analyses attack the same problem, it is an interesting question how they interact. In for example,

$$\begin{array}{l}\texttt{let}\ x = 1 + 2\\ \texttt{in}\ (\lambda y.y + 1)\ x\end{array}$$

both strictness analysis and sharing analysis can be used by turning the binding of x into a strict binding or into a non-updating binding respectively. In these situations the former is usually the better since it also reduces the cost of building closures. The advantage of sharing analysis is that it applies to many situations where strictness analysis does not. Moreover, in the presence of recursive data structures it is difficult to beneficially exploit strictness analysis. For example the *map* function is strict in the spine of its second argument if it is called in a context which requires the spine of the result. To exploit this one could fully evaluate the spine of the second argument to *map* before calling *map*. However, this may be fatal since it can dramatically increase the amount of space needed to run the program. Thus, in practice strictness analysis is used primarily to improve the situation for base types and nonrecurive data types [JP93]. In contrast we believe that recursive data structures do not pose any problem for our sharing analysis.

In general it seems to be a good idea to first apply a strictness analyser and then a sharing analyser. This was the approach taken by Marlow [Mar93] and Peyton Jones and Partain [JP93] and their measurements show that strictness analysis and sharing analysis complement each other .

An interesting observation is that strictness analysis wants to determine whether an expression is used at least ones (in every terminating computation)

while sharing analysis can be considered as determining whether an expression is used at most ones [1]. This suggests that there are important similarities between these analyses. Wright has demonstrated that a strictness logic and a usage logic can be expressed in a common framework [Wri96].

[1] In this work we do not take this approach but rather seek to determine whether references to bindings are used in a linear fashion. That is, whether a reference is duplicated or not. This is important since it guarantees that no dangling pointers are ever created. See section 7.1 for a discussion of this.

Chapter 8

Conclusions and future work

8.1 Conclusions

We have presented a type based sharing analysis that can determine when updates and update marker checks can be avoided. We have proved our analysis sound with respect to the lazy Krivine machine by proving that evaluation preserves typings. As a consequence we get that well typed expressions do not go wrong. We have also proved that our type system enjoys a principal typing property and we have given an algorithm that computes the optimal instance of a principal typing. The analysis has been implemented and the preliminary benchmarks indicate that about 59% of the updates and 99% of the update marker checks can be avoided.

8.2 Future work

8.2.1 Polymorphism

The type system presented in this thesis is monomorphic. For the analysis to be used in a realistic setting it is crucial that the type system can be extended to handle a polymorphic language. Wansbrough and Peyton Jones [WJ99b] also takes Turner et al's type system as their starting point and extends it with (among other things) polymorphism. We think that it is straightforward to extend our type system with polymorphism in a similar fashion. However, Wansbrough's and Peyton Jones type system is only polymorphic in the underlying type system and has no notion of annotation polymorphism which we believe may be important. (In contrast to Wansbrough and Peyton Jones who believe the benefits are likely to be small [WJ99b].) We will come back to

annotation polymorphsim shortly.

8.2.2 User defined data types

Another key feature of functional languages is user defined data types. As it stands our analysis only copes with lists. To extend the type system with user defined data types essentially amounts to giving a way to translate ordinary data type declarations into corresponding bare data type declarations. For example,

$$\begin{array}{rcl} \texttt{data}\,\mathrm{List}\;a & = & \mathrm{Nil} \\ & | & \mathrm{Cons}\;a\;(\mathrm{List}\;a) \end{array}$$

is translated into

$$\begin{array}{rcl} \texttt{data}\,\mathrm{List}\;a\;i_0\;j_0\;k_0\;i_1\;j_1\;k_1 & = & \mathrm{Nil} \\ & | & \mathrm{Cons}\;a_{k_0}^{[i_0,j_0]}\;(\mathrm{List}\;a\;i_0\;j_0\;k_0\;i_1\;j_1\;k_1)_{k_1}^{[i_1,j_1]} \end{array}$$

by turning the type arguments of the constructors into binding types and abstracting over the annotation variables we introduce. This yields the bare type for lists that we have used in our type system (although we have written $\texttt{List}\,a_{k_0}^{[i_0,j_0]}\;k_1\;[i_1,j_1]$ rather than $\mathrm{List}\;a\;i_0\;j_0\;k_0\;i_1\;j_1\;k_1$). By following this idea it should be straightforward to extend the type system with user defined data types. However, as pointed out by Wansbrough and Peyton Jones [WJ98], this means that the number of annotation parameters to the data type grows linearly in the size of the data type declaration. Under the assumption that individual data types do not grow with program size this does not influence the complexity of the analysis but it may have the effect that programs which use a large data type become expensive to analyse. To avoid this Wansbrough and Peyton Jones choose a less expressive type language which, for example, cannot express that the spine of a list is used in a linear fashion while the components are used several times. This clearly degrades the accuracy of the analysis and whether the faster analysis is worth the loss of precision is an interesting topic for future work.

8.2.3 Separate compilation

Separate compilation is a notorious problem for this kind of global program analysis. To find the best annotations of a program it is simply necessary to have access to the whole program. One way to deal with separate compilation is to be conservative and if, for example, append is exported from a module we annotate it as follows.

$$\begin{array}{l} \mathit{append} =^{!} \lambda^{[1,\omega]} xs.\lambda^{[0,\omega]} ys.\texttt{case}\;xs\;\texttt{of} \\ \qquad\qquad\qquad\qquad \texttt{nil} \Rightarrow ys \\ \qquad\qquad\qquad\qquad \texttt{cons}\;z\;zs \Rightarrow \texttt{let}\;ws =^{!} \mathit{append}\;zs\;ys \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{in}\;\texttt{cons}^{[0,\omega]}\;z\;ws \end{array}$$

This means that append will be able to take care of any situation but it also means that append puts demands on its arguments. This is reflected in the types that we can assign to it. Unfortunately, due to the lack of annotation polymorphism there is no minimal type which we can assign to it. But maybe the most useful type we can assign to it is the following (assuming that we have ordinary polymorphism).

$$\begin{aligned}\forall a.\ & (\mathtt{List}\, a^{[0,\omega]}_{!}\ \checkmark\ [0,0])^{[0,0]}_{\checkmark} \to^{[0,\omega]} \\ & (\mathtt{List}\, a^{[0,\omega]}_{!}\ !\ [0,\omega])^{[0,\omega]}_{!} \to^{[0,\omega]} \\ & (\mathtt{List}\, a^{[0,\omega]}_{!}\ !\ [0,\omega])^{[0,\omega]}\end{aligned}$$

This allows the result of append to be used in any context (thanks to subtyping) but it unfortunately puts unnecessarily strong requirements on the types of the arguments. If we choose to export this type then unfortunately every place where we use append will be forced to adapt to this type and it is bound to degrade significantly the accuracy of the analysis. It would be desirable if the effect of the conservative annotations on append could be kept local, that is they only effect the efficiency of the append function itself. We think there are good opportunities to achieve this by means of annotation polymorphism and annotation polyvariance.

8.2.4 Annotation polymorphism

As noted above we believe that annotation polymorphism will be important for the accuracy of the analysis in the setting of large programs and separate compilation. It should be straightforward to extend the modified type system of chapter 5 with annotation polymorphism since it already has the principal typing property. We simply add the possibility to generalise over annotation variables that do not occur in the term. Of course this also means that our type schemes will contain a constraint set which constrains the generalised annotation variables. This will have an significant impact on the complexity of the inference algorithm since instantiation means that we duplicate constraint sets which may lead to an explosion in the number of generated constraints. To avoid some of this increased cost it may be necessary to simplify the constraint set when generalising. How to do this is not clear to us but it seems as if some of techniques usde by Mossin may be applicable [Mos97].

If we add polymorphism then an interesting weakness in our type system shows up. Recall our discussion in the section on separate compilation. There we argued that in the presence of separate compilation we will have to be conservative and, for example, annotate append as follows.

$$\begin{aligned}append =^{!} \lambda^{[1,\omega]} xs.\lambda^{[0,\omega]} ys.&\mathtt{case}\ xs\ \mathtt{of} \\ & \mathtt{nil} \Rightarrow ys \\ & \mathtt{cons}\ z\ zs \Rightarrow \mathtt{let}\ ws =^{!} append\ zs\ ys \\ & \qquad\qquad \mathtt{in}\ \mathtt{cons}^{[0,\omega]}\ z\ ws\end{aligned}$$

We also argued that it would be desirable if the effect of these conservative annotations could be kept local. Using the notion of annotation polymorphism suggested above we could assign a type like

$$\begin{array}{ll} \forall a, i_0, i_1, i_2, j_0, j_1, j_2, k_0, k_1. & (\texttt{List}\, a_{!}^{[i_0,j_0]} \checkmark [0,0])_{\checkmark}^{[0,0]} \rightarrow^{[0,\omega]} \\ & (\texttt{List}\, a_{k_0}^{[i_0,j_0]}\, k_1\, [i_1,j_1])_{k_1}^{[i_1,j_1]} \rightarrow^{[i_2,j_2]} \\ & (\texttt{List}\, a_{k_0}^{[i_0,j_0]}\, k_1\, [i_1,j_1])^{[i_1,j_1]} \\ & \text{where} \quad 1 \leq i_1 \\ & \qquad \texttt{if}\ j_2 > 0\ \texttt{then}\ k_1 = ! \end{array}$$

to our example [1]. This type is clearly much better than the type we can assign to it in our system without annotation polymorphism. However it is not entirely satisfactory. For example the type demands that the elements of the list of the first argument to append get updated. This is due to the fact that we have annotated the binding of ws with ! which allows the spine of the list returned by append to be used several times. The type system then also enforces that the elements of the list given as the first argument can be used several times since they end up as elements in the resulting list. The constraint $1 \leq i_1$ will have a similar effect on the elements of the second argument since it will in force k_0 and k_1 to be ! (unless we just pass append the empty list). It is however not necessary for the elements of the lists to be updating if we use the result of append in a context where it is used linearly and we would expect this to be reflected in the type. To explain why it is not so, consider the following simple example.

$$\begin{array}{l} \texttt{let}\, x =^{\checkmark} 1 +^{[0,0]} 2 \\ \quad\ \ y =^{\checkmark} 3 +^{[0,0]} 4 \\ \quad\ \ f =^{!} \lambda^{[1,1]} z.y +^{[0,0]} z \\ \texttt{in}\, f\ x \end{array}$$

This is ill-typed but well-annotated (ie it does not go wrong). It is ill-typed since f is annotated with ! and therefore we will duplicate $\lambda z.y + z$ when we look up f and thus also duplicate y. Since we duplicate y we will give it the type $\texttt{Int}_{!}^{[0,0]}$ which clashes with the annotation on the binding for y. It is well-annotated since the duplication is completely harmless: when f is looked up the binding for f becomes garbage; we may remove it and there will still only be one occurrence of y. Our type system can not express this and we think it necessary to use a richer type language to do so. It should be noted that this refinement is only important in the presence of large programs and separate compilation since

[1] We could actually assign it an even better but more complicated type. For clarity we have simplified away some subtyping constraints.

when we have access to the whole program we would annotate the example as

$$\begin{array}{l}\texttt{let}\ x =^{\checkmark} 1 +^{[0,0]} 2\\ \quad\ \ y =^{\checkmark} 3 +^{[0,0]} 4\\ \quad\ \ f =^{\checkmark} \lambda^{[0,0]} z.y +^{[0,0]} z\\ \texttt{in}\ f\ x\end{array}$$

which is well-typed and optimally annotated. In the presence of separate compilation we will, as noted above, sometimes need to be unnecessary conservative and not annotate the term optimally and thus we think that this may be important for the overall accuracy of the analysis.

8.2.5 Annotation polyvariance

Even if we add annotation polymorphism and make our type system more parametric the effect of conservative annotations cannot be kept entirely local. However it may be possible to achieve this locality by generating a few differently annotated versions of every function we export. We would do that by carefully generating a few versions such that the set of types that we can assign to them coincides with the set of types that we could assign to the function if we could generate all possible versions (which we cannot do since that is infinitely many). We would, for example, generate at least two versions of append, one that builds lists that can be shared and one that builds non-sharable lists. Whether this is actually possible is not clear to us and it is an interesting topic for future work.

Another simpler form of polyvariance can be used if a function is used in several different contexts within the same module. Then one can simply generate one version of each function for every use.

8.2.6 Garbage collection of update markers

As discussed in section 2.4 our analysis precludes that update markers are garbage collected. Some compilers do not garbage collect update markers and that does not seem to cause any problems in practice. Anyhow, there are programs which run in constant stack space with garbage collection of update markers but which may run out of stack without it. See section 2.4 for an example of this. Fortunately, there seems to be a solution at hand. If all bindings in a program can be assigned a type of the form $\rho_\kappa^{[\eta,\xi]}$ where $\xi \neq \omega$ then we know that arbitray many update markers cannot be stacked on top of each other. This means that the amount of stack needed by the program in an implementation without garbage collection of update markers is within a constant factor of the stack space needed in an implementation with it (and an upper bound on the constant factor is given by the ξ's). Most programs seem to fall in this category. Conversely, if any binding in the program is assigned a type of the form $\rho_\kappa^{[\eta,\omega]}$ then arbitrarily many update markers may stack up on top of each other. In those examples we may simply turn off the analysis and garbage collect update

markers as usual. A more sophisticated, but more complicated, solution would be to have two different sorts of update markers. One that may be garbage collected and one that may not. We could then adapt the analysis such that an arbitrary number of update markers of the latter sort do not stack up on top of each other.

8.2.7 The implementations of abstract machines

The benchmarks presented in chapter 6 suggest that checking for update markers is an operation which is about ten times as frequent as the pushing and popping of update markers. It is therefore no surprise that implementations of abstract machines tend to be optimised towards fast update marker checks at the expense of the pushing and popping of update markers. However, with our analysis update marker checks can be avoided to such an extent that they become far less frequent than the pushing and popping of update markers. Hence, with our analysis, the implementation of the abstract machine should instead be optimised towards fast pushing and popping of update markers at the expense of the update marker checks and we note this as an interesting topic to explore.

8.2.8 A possibility for further analysis

It is often the case that one optimisation opens up possibilities for further optimisations. Indeed, this is true for our analysis. Due to our analysis most update markers are never checked for (since most checks can be avoided). A marker that is never checked for could be represented by just the pointer of where in the heap to update. This kind of marker would be very cheap to push and pop. We are currently working on an analysis for this purpose, that is an analysis which can ensure that a marker is never checked for. We believe that the analysis could be very effective and would allow us to represent most markers in this cheap way. Indeed this would make the tradeoff discussed in the previous subsection less of an issue since it would only concern the remaining markers that might need to be checked.

8.2.9 The analysis in an optimising compiler

The initial experimental results given in chapter 6 seem very promising but indeed the programs tried out are very small. It would be interesting to incorporate the analysis into an optimising compiler and measure the effectiveness of the analysis in terms of the actual speedup of large real world programs.

Appendix A

Typing rules

In this appendix we have collected all the typing rules to provide a convenient overview of the rules. Figures A.1 to A.5 contain the typing rules for the type system presented in chapter 3. Figures A.6 to A.8 contain the rules for typing configurations presented in chapter 4. Figure A.9 contains the typing rules of the underlying ordinary type system and figures A.10 to A.12 contain the typing rules for the modified type system presented in chapter 5.

$$\text{Abs}\frac{\Gamma_0, \Gamma_1 \vdash e \rightsquigarrow \tilde{e} : \tau}{\Gamma_0 \vdash \lambda x.e \rightsquigarrow \lambda x.\tilde{e} : \sigma \to \tau} \quad \begin{array}{l} x \notin \mathrm{dom}(\Gamma_0) \\ x : \sigma \to^* \Gamma_1 \end{array}$$

$$\text{Int}\frac{}{\vdash n \rightsquigarrow n : \mathtt{Int}} \qquad \text{Nil}\frac{}{\vdash \mathtt{nil} \rightsquigarrow \mathtt{nil} : \mathtt{List}\,\sigma\,\kappa\,\iota}$$

$$\text{Cons}\frac{}{x : \sigma_0,\ y : \sigma_1 \vdash \mathtt{cons}\,x\,y \rightsquigarrow \mathtt{cons}\,x\,y : \mathtt{List}\,\sigma\,\kappa\,\iota} \quad \begin{array}{l} \sigma_0 \leq \sigma \\ \sigma_1 \leq (\mathtt{List}\,\sigma\,\kappa\,\iota)^{\iota}_{\kappa} \end{array}$$

Figure A.1: Typing rules for bare values

$$\text{Value}\frac{\Gamma \vdash v \rightsquigarrow \tilde{v} : \rho}{\Gamma \vdash v \rightsquigarrow \tilde{v}^{\iota} : \rho^{[\eta,\xi]}} \quad \begin{array}{l}\text{if } \xi > 0 \text{ then } \Gamma \to^* \Gamma, \Gamma \\ [\eta, \xi] \leq \iota\end{array}$$

$$\text{Var}\frac{}{x : \tau_{\kappa} \vdash x \rightsquigarrow x : \tau'} \quad \tau \leq \tau' \qquad \text{App}\frac{\Gamma \vdash e \rightsquigarrow \tilde{e} : \sigma \to^{[0,0]} \tau}{\Gamma,\ x : \sigma \vdash e\ x \rightsquigarrow \tilde{e}\ x : \tau}$$

$$\text{Plus}\frac{\Gamma_0 \vdash e_0 \rightsquigarrow \tilde{e}_0 : \texttt{Int}^{[0,0]} \qquad \Gamma_1 \vdash e_1 \rightsquigarrow \tilde{e}_1 : \texttt{Int}^{[0,0]}}{\Gamma_0,\ \Gamma_1 \vdash e_0 + e_1 \rightsquigarrow \tilde{e}_0 +^{\iota} \tilde{e}_1 : \texttt{Int}^{\iota'}} \quad \iota' \leq \iota$$

$$\text{Add}\frac{\Gamma \vdash e \rightsquigarrow \tilde{e} : \texttt{Int}^{[0,0]}}{\Gamma \vdash \texttt{add}_n\ e \rightsquigarrow \texttt{add}^{\iota}_n\ \tilde{e} : \texttt{Int}^{\iota'}} \quad \iota' \leq \iota$$

$$\text{Let}\frac{\Gamma_0,\ \Gamma_1 \vdash d \rightsquigarrow \tilde{d} : \Delta \qquad \Gamma_2,\ \Gamma_3 \vdash e \rightsquigarrow \tilde{e} : \tau}{\Gamma_0,\ \Gamma_2 \vdash \texttt{let}\ d\ \texttt{in}\ e \rightsquigarrow \texttt{let}\ \tilde{d}\ \texttt{in}\ \tilde{e} : \tau} \quad \begin{array}{l}\text{dom}(\tilde{d}) \cap \text{dom}(\Gamma_0, \Gamma_2) = \emptyset \\ \Delta \vdash \Gamma_1,\ \Gamma_3\end{array}$$

$$\text{Case}\frac{\Gamma_0 \vdash e \rightsquigarrow \tilde{e} : \rho^{[0,0]} \qquad \Gamma_1 \vdash alts \rightsquigarrow \tilde{alts} : \rho \Rightarrow \tau}{\Gamma_0,\ \Gamma_1 \vdash \texttt{case}\ e\ \texttt{of}\ alts \rightsquigarrow \texttt{case}\ \tilde{e}\ \texttt{of}\ \tilde{alts} : \tau}$$

Figure A.2: Typing rules for expressions

$$\text{Bind-}\checkmark\ \frac{\Gamma \vdash e \rightsquigarrow \tilde{e} : \tau}{\Gamma \vdash x = e \rightsquigarrow x =^{\checkmark} \tilde{e} : (x : \tau_{\checkmark})}$$

$$\text{Bind-!}\frac{\Gamma \vdash e \rightsquigarrow \tilde{e} : \rho^{[\eta+1,\xi+1]}}{\Gamma \vdash x = e \rightsquigarrow x =^{!} \tilde{e} : (x : \rho_{\kappa}^{[\eta,\xi]})}$$

Figure A.3: Typing rules for bindings

$$\text{Decl-}\epsilon\frac{}{\vdash \epsilon \rightsquigarrow \epsilon : \epsilon}$$

$$\text{Decl}\frac{\Gamma_0 \vdash d \rightsquigarrow \tilde{d} : \Delta \qquad \Gamma_1 \vdash b \rightsquigarrow \tilde{b} : (x : \sigma)}{\Gamma_0, \Gamma_1 \vdash d,\ b \rightsquigarrow \tilde{d},\ \tilde{b} : (\Delta, x : \sigma)}$$

Figure A.4: Typing rules for declarations

$$\text{Alts}\frac{\Gamma_0, \Gamma_1 \vdash e_0 \rightsquigarrow \tilde{e}_0 : \tau \qquad \Gamma_0, \Gamma_2, \Gamma_3 \vdash e_1 \rightsquigarrow \tilde{e}_1 : \tau}{\begin{array}{l}\Gamma_0, \Gamma_1, \Gamma_2 \vdash \{\texttt{nil} \Rightarrow e_0; \texttt{cons}\, x\, y \Rightarrow e_1\} \rightsquigarrow \\ \qquad \{\texttt{nil} \Rightarrow \tilde{e}_0; \texttt{cons}\, x\, y \Rightarrow \tilde{e}_1\} : \texttt{List}\, \sigma\, \kappa\, \iota \Rightarrow \tau\end{array}} \quad \begin{array}{l} x, y \notin \mathrm{dom}(\Gamma_0, \Gamma_2) \\ x : \sigma, \\ y : (\texttt{List}\, \sigma\, \kappa\, \iota)^{\iota}_{\kappa} \end{array} \vdash \Gamma_3$$

Figure A.5: Typing rule for alternatives

$$\text{AppR}\frac{}{x : \sigma \vdash [\,]\, x \rightsquigarrow [\,]\, x : [\sigma \to^{[0,0]} \tau]\tau}$$

$$\text{PlusR}\frac{\Gamma \vdash e \rightsquigarrow \tilde{e} : \texttt{Int}^{[0,0]}}{\Gamma \vdash [\,] + e \rightsquigarrow [\,] +^{\iota} \tilde{e} : [\texttt{Int}^{[0,0]}]\texttt{Int}^{\iota'}} \quad \iota' \leq \iota$$

$$\text{AddR}\frac{}{\vdash \texttt{add}_n\, [\,] \rightsquigarrow \texttt{add}^{\iota}_n\, [\,] : [\texttt{Int}^{[0,0]}]\texttt{Int}^{\iota'}} \quad \iota' \leq \iota$$

$$\text{CaseR}\frac{\Gamma \vdash alts \rightsquigarrow \tilde{alts} : \rho \Rightarrow \tau}{\Gamma \vdash \texttt{case}\, [\,]\, \texttt{of}\, alts \rightsquigarrow \texttt{case}\, [\,]\, \texttt{of}\, \tilde{alts} : [\rho^{[0,0]}]\tau}$$

Figure A.6: Typing rules for reduction contexts

$$\text{Stack-}\epsilon\frac{}{\vdash \epsilon \rightsquigarrow \epsilon : \epsilon\,;\, [\tau]\tau}$$

$$\text{Stack-}R\frac{\Gamma_0 \vdash R \rightsquigarrow \tilde{R} : [\rho^{[0,0]}]\tau_0 \qquad \Gamma_1 \vdash S \rightsquigarrow \tilde{S} : \Delta\,;\, [\tau_0]\tau_1}{\Gamma_0,\, \Gamma_1 \vdash R, S \rightsquigarrow \tilde{R}, \tilde{S} : \Delta\,;\, [\rho^{[0,0]}]\tau_1}$$

$$\text{Stack-}\#\frac{\Gamma \vdash S \rightsquigarrow \tilde{S} : \Delta\,;\, [\tau_0]\tau_1}{\Gamma \vdash \#x,\, S \rightsquigarrow \#x,\, \tilde{S} : (\Delta,\, x : \rho^{[\eta,\xi]}_{\kappa}\,;\, [\rho^{[\eta+1,\xi+1]}]\tau_1)} \quad \rho^{[\eta,\xi]} \leq \tau_0$$

$$\text{Stack-}\#\text{-discard}\frac{\Gamma \vdash S \rightsquigarrow \tilde{S} : \Delta\,;\, [\tau_0]\tau_1}{\Gamma \vdash \#x,\, S \rightsquigarrow \tilde{S} : \Delta\,;\, [\tau_2]\tau_1} \quad \tau_2 \leq \tau_0$$

Figure A.7: Typing rules for stacks

$$\text{Config}\frac{\begin{array}{l}\Gamma_0 \vdash H \rightsquigarrow \tilde{H} : \Delta_0 \\ \Gamma_1 \vdash e \rightsquigarrow \tilde{e} : \tau_0 \\ \Gamma_2 \vdash S \rightsquigarrow \tilde{S} : \Delta_1 \,;\, [\tau_0]\tau_1\end{array}}{\vdash \langle H \,;\, e \,;\, S\rangle \rightsquigarrow \langle \tilde{H} \,;\, \tilde{e} \,;\, \tilde{S}\rangle : \tau_1} \quad \Delta_0,\, \Delta_1 \vdash \Gamma_0,\, \Gamma_1,\, \Gamma_2$$

Figure A.8: Typing rule for configurations

$$\text{Int}\frac{}{\Omega \vdash n : \texttt{Int}} \qquad \text{Nil}\frac{}{\Omega \vdash \texttt{nil} : \texttt{List}\,\delta} \qquad \text{Cons}\frac{\Omega(x) = \delta \qquad \Omega(y) = \texttt{List}\,\delta}{\Omega \vdash \texttt{cons}\,x\,y : \texttt{List}\,\delta}$$

$$\text{Abs}\frac{\Omega,\, x : \delta_0 \vdash \underline{e} : \delta_1}{\Omega \vdash \lambda x.\underline{e} : \delta_0 \to \delta_1} \qquad \text{Var}\frac{\Omega(x) = \delta}{\Omega \vdash x : \delta} \qquad \text{Plus}\frac{\Omega \vdash \underline{e}_0 : \texttt{Int} \qquad \Omega \vdash \underline{e}_1 : \texttt{Int}}{\Omega \vdash \underline{e}_0 + \underline{e}_1 : \texttt{Int}}$$

$$\text{Add}\frac{\Omega \vdash \underline{e} : \texttt{Int}}{\Omega \vdash \texttt{add}_n\,\underline{e} : \texttt{Int}} \qquad \text{App}\frac{\Omega \vdash \underline{e} : \delta_0 \to \delta_1 \qquad \Omega(x) = \delta_0}{\Omega \vdash \underline{e}\,x : \delta_1}$$

$$\text{Case}\frac{\Omega \vdash \underline{e} : \delta_1 \qquad \Omega \vdash \underline{\mathit{alts}} : \delta_1 \Rightarrow \delta_0}{\Omega \vdash \texttt{case}\,\underline{e} : \delta_1\,\texttt{of}\,\underline{\mathit{alts}} : \delta_0}$$

$$\text{Let}\frac{\Omega_0,\, \Omega_1 \vdash \underline{d} : \Omega_1 \qquad \Omega_0,\, \Omega_1 \vdash \underline{e} : \delta}{\Omega_0 \vdash \texttt{let}\,\underline{d} : \Omega_1\,\texttt{in}\,\underline{e} : \delta}$$

$$\text{Alts}\frac{\Omega \vdash \underline{e}_0 : \delta_0 \qquad \Omega,\, x : \delta_1,\, y : \texttt{List}\,\delta_1 \vdash \underline{e}_1 : \delta_0}{\Omega \vdash \{\texttt{nil} \Rightarrow \underline{e}_0; \texttt{cons}\,x\,y \Rightarrow \underline{e}_1\} : \texttt{List}\,\delta_1 \Rightarrow \delta_0}$$

$$\text{Declaration-}\epsilon\frac{}{\Omega \vdash \epsilon : \epsilon} \qquad \text{Declaration}\frac{\Omega \vdash \underline{d} : \Omega_0 \qquad \Omega \vdash \underline{b} : (x : \delta)}{\Omega \vdash \underline{d},\, \underline{b} : (\Omega_0, x : \delta)}$$

$$\text{Binding}\frac{\Omega \vdash \underline{e} : \delta}{\Omega \vdash x = \underline{e} : (x : \delta)}$$

Figure A.9: Ordinary typing rules

$$\text{Int}\frac{}{\Pi\,;\ \epsilon \vdash n \rightsquigarrow n : \texttt{Int}} \qquad \text{Nil}\frac{}{\Pi\,;\ \epsilon \vdash \texttt{nil} \rightsquigarrow \texttt{nil} : \texttt{List}\ \sigma\ \kappa\ \iota}$$

$$\text{Cons}\frac{}{\Pi\,;\ x : \sigma_0, y : \sigma_1 \vdash \texttt{cons}\ x\ y \rightsquigarrow \texttt{cons}\ x\ y : \texttt{List}\ \sigma\ \kappa\ \iota} \quad \begin{array}{l}\Pi \vdash \sigma_0 \leq \sigma \\ \Pi \vdash \sigma_1 \leq (\texttt{List}\ \sigma\ \kappa\ \iota)^{\iota}_{\kappa}\end{array}$$

$$\text{Abs}\frac{\Pi\,;\ \Gamma_0, \Gamma_1 \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau}{\Pi\,;\ \Gamma_0 \vdash \lambda x.\underline{e} \rightsquigarrow \lambda x.\underline{\tilde{e}} : \sigma \to \tau} \quad \begin{array}{l} x \notin \mathrm{dom}(\Gamma_0) \\ \Pi \vdash x : \sigma \to^* \Gamma_1\end{array}$$

Figure A.10: Typing rules for bare values

$$\text{Value}\frac{\Pi\,;\ \Gamma \vdash \underline{v} \rightsquigarrow \underline{\tilde{v}} : \rho}{\Pi\,;\ \Gamma \vdash \underline{v} \rightsquigarrow \underline{\tilde{v}}^{\iota} : \rho^{[\eta,\xi]}} \quad \begin{array}{l}\Pi \vdash \texttt{if}\ \xi > 0\ \texttt{then}\ \Gamma \to^* \Gamma, \Gamma \\ \Pi \vdash [\eta, \xi] \leq \iota\end{array}$$

$$\text{Var}\frac{}{\Pi\,;\ x : \tau_{\kappa} \vdash x \rightsquigarrow x : \tau'} \quad \Pi \vdash \tau \leq \tau'$$

$$\text{Plus}\frac{\Pi\,;\ \Gamma_0 \vdash \underline{e}_0 \rightsquigarrow \underline{\tilde{e}}_0 : \texttt{Int}^{[0,0]} \qquad \Pi\,;\ \Gamma_1 \vdash \underline{e}_1 \rightsquigarrow \underline{\tilde{e}}_1 : \texttt{Int}^{[0,0]}}{\Pi\,;\ \Gamma_0, \Gamma_1 \vdash \underline{e}_0 + \underline{e}_1 \rightsquigarrow \underline{\tilde{e}}_0 +^{\iota} \underline{\tilde{e}}_1 : \texttt{Int}^{\iota'}} \quad \Pi \vdash \iota' \leq \iota$$

$$\text{Add}\frac{\Pi\,;\ \Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \texttt{Int}^{[0,0]}}{\Pi\,;\ \Gamma \vdash \texttt{add}_n\ \underline{e} \rightsquigarrow \texttt{add}^{\iota}_n\ \underline{\tilde{e}} : \texttt{Int}^{\iota'}} \quad \Pi \vdash \iota' \leq \iota$$

$$\text{App}\frac{\Pi\,;\ \Gamma \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \sigma \to^{[0,0]} \tau}{\Pi\,;\ \Gamma, x : \sigma \vdash \underline{e}\ x \rightsquigarrow \underline{\tilde{e}}\ x : \tau}$$

$$\text{Case}\frac{\Pi\,;\ \Gamma_0 \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \rho^{[0,0]} \qquad \Pi\,;\ \Gamma_1 \vdash \underline{alts} \rightsquigarrow \underline{\tilde{alts}} : \rho \Rightarrow \tau}{\Pi\,;\ \Gamma_0, \Gamma_1 \vdash \texttt{case}\ \underline{e} : \delta\ \texttt{of}\ \underline{alts} \rightsquigarrow \texttt{case}\ \underline{\tilde{e}} : \rho\ \texttt{of}\ \underline{\tilde{alts}} : \tau} \quad \lceil \rho \rceil \equiv \delta$$

$$\text{Let}\frac{\Pi\,;\ \Gamma_0, \Gamma_1 \vdash \underline{d} \rightsquigarrow \underline{\tilde{d}} : \Delta \qquad \Pi\,;\ \Gamma_2, \Gamma_3 \vdash \underline{e} \rightsquigarrow \underline{\tilde{e}} : \tau}{\Pi\,;\ \Gamma_0, \Gamma_2 \vdash \texttt{let}\ \underline{d} : \Phi\ \texttt{in}\ \underline{e} \rightsquigarrow \texttt{let}\ \underline{\tilde{d}} : \Delta\ \texttt{in}\ \underline{\tilde{e}} : \tau} \quad \begin{array}{l}\mathrm{dom}(\underline{\tilde{d}}) \cap \mathrm{dom}(\Gamma_0, \Gamma_2) = \emptyset \\ \Pi \vdash (\Delta \vdash \Gamma_1, \Gamma_3) \\ \lceil \Delta \rceil \equiv \Phi\end{array}$$

Figure A.11: Typing rules

$$\text{Alts}\frac{\Pi\,;\,\Gamma_0,\Gamma_1 \vdash \underline{e}_0 \rightsquigarrow \tilde{e}_0 : \tau \qquad \Pi\,;\,\Gamma_0,\Gamma_2,\Gamma_3 \vdash \underline{e}_1 \rightsquigarrow \tilde{e}_1 : \tau}{\begin{array}{l}\Pi\,;\,\Gamma_0,\Gamma_1,\Gamma_2 \vdash \{\mathtt{nil} \Rightarrow \underline{e}_0; \mathtt{cons}\ x\ y \Rightarrow \underline{e}_1\} \rightsquigarrow \\ \qquad\qquad \{\mathtt{nil} \Rightarrow \tilde{e}_0; \mathtt{cons}\ x\ y \Rightarrow \tilde{e}_1\} : \mathtt{List}\ \sigma\ \kappa\ \iota \Rightarrow \tau\end{array}}$$
where $x, y \notin \mathrm{dom}(\Gamma_0, \Gamma_2)$
$\Pi \vdash x : \sigma, y : (\mathtt{List}\ \sigma\ \kappa\ \iota)^{\iota}_{\kappa} \vdash \Gamma_3$

Figure A.12: Typing rule for alternatives

$$\text{Decl-}\epsilon\frac{}{\Pi\,;\,\epsilon \vdash \epsilon \rightsquigarrow \epsilon : \epsilon}$$

$$\text{Decl}\frac{\Pi\,;\,\Gamma_0 \vdash \underline{d} \rightsquigarrow \tilde{\underline{d}} : \Delta \qquad \Pi\,;\,\Gamma_1 \vdash \underline{b} \rightsquigarrow \tilde{\underline{b}} : (x : \sigma)}{\Pi\,;\,\Gamma_0,\Gamma_1 \vdash \underline{d},\ \underline{b} \rightsquigarrow \tilde{\underline{d}},\ \tilde{\underline{b}} : (\Delta, x : \sigma)}$$

$$\text{Bind}\frac{\Pi\,;\,\Gamma \vdash \underline{e} \rightsquigarrow \tilde{\underline{e}} : \rho^{[\eta\oplus\kappa, \xi\oplus\kappa]}}{\Pi\,;\,\Gamma \vdash x = \underline{e} \rightsquigarrow x =^{\kappa} \tilde{\underline{e}} : (x : \rho^{[\eta,\xi]}_{\kappa'})} \quad \kappa' \leq \kappa \in \Pi$$

Figure A.13: Typing rules for declarations and bindings

Appendix B

Constraint inference algorithm

This appendix contains the full definition of the constraint inference algorithm described in subsection 5.5.2 which is an important part of the type inference algorithm described in chapter 5.

$$
\begin{array}{lll}
\texttt{infer}(\Delta, n, \texttt{Int}) = (\emptyset, \epsilon) & & \\
\texttt{infer}(\Delta, \texttt{nil}, \texttt{List}\ \sigma\ \kappa\ \iota) = (\emptyset, \epsilon) & & \\
\texttt{infer}(\Delta, \texttt{cons}\ x\ y, \texttt{List}\ \sigma\ \kappa\ \iota) = & (\Pi, \Gamma) & \\
 & \text{where} & \sigma_0 = \Delta(x) \\
 & & \sigma_1 = \Delta(y) \\
 & & \Pi_0 = \texttt{infer}(\sigma_0 \leq \sigma) \\
 & & \Pi_1 = \texttt{infer}(\sigma_1 \leq (\texttt{List}\ \sigma\ \kappa\ \iota)^{\iota}_{\kappa}) \\
 & & \Pi = \Pi_0, \Pi_1 \\
 & & \Gamma = x : \sigma_0, y : \sigma_1 \\
\texttt{infer}(\Delta_0, \lambda x.\underline{\tilde{e}}, \sigma \to \tau) = & (\Pi, \Gamma) & \\
 & \text{where} & \Delta_1 = \Delta_0, x : \sigma \\
 & & (\Pi_0, \Gamma_0) = \texttt{infer}(\Delta_1, \underline{\tilde{e}}, \tau) \\
 & & \Pi_1 = \texttt{infer}(x : \sigma \to^* \Gamma_2) \\
 & & \Pi = \Pi_0, \Pi_1 \\
 & & \Gamma = \Gamma_1 \\
 & & \text{where}\ \Gamma_1, \Gamma_2 \equiv \Gamma_0 \\
 & & \quad x \notin \mathrm{dom}(\Gamma_1) \\
 & & \quad \mathrm{dom}(\Gamma_2) \subseteq \{x\}
\end{array}
$$

Figure B.1: Definition of $\texttt{infer}(\Delta, \underline{\tilde{v}}, \rho)$

$$
\begin{array}{l}
\texttt{infer}(\Delta, \tilde{v}^{\iota}, \rho^{[\eta,\xi]}) = \ (\Pi, \Gamma) \\
\qquad \text{where } (\Pi_0, \Gamma) = \texttt{infer}(\Delta, \tilde{v}, \rho) \\
\qquad\qquad \Pi_1 = \texttt{infer}(\texttt{if } \xi > 0 \texttt{ then } \Gamma \rightarrow^{*} \Gamma, \Gamma) \\
\qquad\qquad \Pi_2 = \texttt{infer}([\eta, \xi] \leq \iota) \\
\qquad\qquad \Pi = \Pi_0, \Pi_1, \Pi_2 \\
\texttt{infer}(\Delta, x, \tau) = \ (\Pi, \Gamma) \\
\qquad \text{where } \tau'_{\kappa} = \Delta(x) \\
\qquad\qquad \Pi = \texttt{infer}(\tau' \leq \tau) \\
\qquad\qquad \Gamma = x : \tau' \\
\texttt{infer}(\Delta, \underline{\tilde{e}}_0 +^{\iota} \underline{\tilde{e}}_1, \texttt{Int}^{\iota'}) = \ (\Pi, \Gamma) \\
\qquad\qquad \text{where } (\Pi_0, \Gamma_0) = \texttt{infer}(\Delta, \underline{\tilde{e}}_0, \texttt{Int}^{[0,0]}) \\
\qquad\qquad\qquad (\Pi_1, \Gamma_1) = \texttt{infer}(\Delta, \underline{\tilde{e}}_1, \texttt{Int}^{[0,0]}) \\
\qquad\qquad\qquad \Pi_2 = \texttt{infer}(\iota' \leq \iota) \\
\qquad\qquad\qquad \Pi = \Pi_0, \Pi_1, \Pi_2 \\
\qquad\qquad\qquad \Gamma = \Gamma_0, \Gamma_1 \\
\texttt{infer}(\Delta, \texttt{add}^{\iota}_{n}\, \underline{\tilde{e}}, \texttt{Int}^{\iota'}) = \ (\Pi, \Gamma) \\
\qquad\qquad \text{where } (\Pi_0, \Gamma) = \texttt{infer}(\Delta, \underline{\tilde{e}}, \texttt{Int}^{[0,0]}) \\
\qquad\qquad\qquad \Pi_1 = \texttt{infer}(\iota' \leq \iota) \\
\qquad\qquad\qquad \Pi = \Pi_0, \Pi_1 \\
\texttt{infer}(\Delta, \underline{\tilde{e}}\ x, \tau) = \ (\Pi, \Gamma) \\
\qquad \text{where } \sigma = \Delta(x) \\
\qquad\qquad (\Pi, \Gamma_0) = \texttt{infer}(\Delta, \underline{\tilde{e}}, \sigma \rightarrow^{[0,0]} \tau) \\
\qquad\qquad \Gamma = \Gamma_0, x : \sigma \\
\texttt{infer}(\Delta, \texttt{case } \underline{\tilde{e}} : \rho \texttt{ of } \underline{\tilde{alts}}, \tau) = \ (\Pi, \Gamma) \\
\qquad\qquad\qquad \text{where } (\Pi_0, \Gamma_0) = \texttt{infer}(\Delta, \underline{\tilde{e}}, \rho^{[0,0]}) \\
\qquad\qquad\qquad\qquad (\Pi_1, \Gamma_1) = \texttt{infer}(\Delta, \underline{\tilde{alts}}, \rho \Rightarrow \tau) \\
\qquad\qquad\qquad\qquad \Pi = \Pi_0, \Pi_1 \\
\qquad\qquad\qquad\qquad \Gamma = \Gamma_0, \Gamma_1 \\
\texttt{infer}(\Delta_0, \texttt{let } \underline{\tilde{d}} : \Delta_1 \texttt{ in } \underline{\tilde{e}}, \tau) = \ (\Pi, \Gamma) \\
\qquad\qquad\qquad\qquad \text{where } \Delta = \Delta_0, \Delta_1 \\
\qquad\qquad\qquad\qquad\qquad (\Pi_0, \Gamma_0) = \texttt{infer}(\Delta, \underline{\tilde{d}}, \Delta_1) \\
\qquad\qquad\qquad\qquad\qquad (\Pi_1, \Gamma_1) = \texttt{infer}(\Delta, \underline{\tilde{e}}, \tau) \\
\qquad\qquad\qquad\qquad\qquad \Pi_2 = \texttt{infer}(\Delta_1 \vdash \Gamma_3, \Gamma_5) \\
\qquad\qquad\qquad\qquad\qquad \Pi = \Pi_0, \Pi_1, \Pi_2 \\
\qquad\qquad\qquad\qquad\qquad \Gamma = \Gamma_2, \Gamma_4 \\
\qquad\qquad\qquad\qquad\qquad \text{where } \Gamma_2, \Gamma_3 \equiv \Gamma_0 \\
\qquad\qquad\qquad\qquad\qquad\qquad \Gamma_4, \Gamma_5 \equiv \Gamma_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{dom}(\underline{\tilde{d}}) \cap \mathrm{dom}(\Gamma_2, \Gamma_4) = \emptyset \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{dom}(\Gamma_3, \Gamma_5) \subseteq \mathrm{dom}(\underline{\tilde{d}})
\end{array}
$$

Figure B.2: Definition of $\texttt{infer}(\Delta, \underline{\tilde{e}}, \tau)$

$$\begin{array}{l}\texttt{infer}(\Delta_0, \{\texttt{nil} \Rightarrow \underline{\tilde{e}}_0; \texttt{cons}\ x\ y \Rightarrow \underline{\tilde{e}}_1\}, \texttt{List}\,\sigma\,\kappa\,\iota \Rightarrow \tau) = (\Pi, \Gamma) \\ \qquad \text{where } (\Pi_0, \Gamma_0) = \texttt{infer}(\Delta_0, \underline{\tilde{e}}_0, \tau) \\ \qquad\quad \Delta_1 = x : \sigma, y : (\texttt{List}\,\sigma\,\kappa\,\iota)^{\iota}_{\kappa} \\ \qquad\quad \Delta_2 = \Delta_0, \Delta_1 \\ \qquad\quad (\Pi_1, \Gamma_1) = \texttt{infer}(\Delta_2, \underline{\tilde{e}}_1, \tau) \\ \qquad\quad \Pi_2 = \texttt{infer}(\Delta_1 \vdash \Gamma_5) \\ \qquad\quad \Pi = \Pi_0, \Pi_1, \Pi_2 \\ \qquad\quad \Gamma = \Gamma_2, \Gamma_3, \Gamma_4 \\ \qquad\quad \text{where } \Gamma_2, \Gamma_3 \equiv \Gamma_0 \\ \qquad\qquad \Gamma_2, \Gamma_4, \Gamma_5 \equiv \Gamma_1 \\ \qquad\qquad x, y \notin \mathrm{dom}(\Gamma_2, \Gamma_4) \\ \qquad\qquad \mathrm{dom}(\Gamma_5) \subseteq \{x, y\} \\ \qquad\qquad \Gamma_3 \cap \Gamma_4 = \epsilon \end{array}$$

Figure B.3: Definition of $\texttt{infer}(\Delta, \underline{\tilde{alts}}, \rho \Rightarrow \tau)$

$$\begin{array}{l}\texttt{infer}(\Delta, \epsilon, \epsilon) = (\emptyset, \epsilon) \\ \texttt{infer}(\Delta_0, \underline{\tilde{d}}_0, \Delta_1) = \quad (\Pi, \Gamma) \\ \qquad\qquad \text{where } (\Pi_0, \Gamma_0) = \texttt{infer}(\Delta_0, \underline{\tilde{d}}_1, \Delta_2) \\ \qquad\qquad\quad (\Pi_1, \Gamma_1) = \texttt{infer}(\Delta_0, x =^{\kappa} \underline{\tilde{e}}, x : \sigma) \\ \qquad\qquad\quad \Pi = \Pi_0, \Pi_1 \\ \qquad\qquad\quad \Gamma = \Gamma_1, \Gamma_2 \\ \qquad\qquad\quad \text{where } \Delta_2, x : \sigma \equiv \Delta_1 \\ \qquad\qquad\qquad \underline{\tilde{d}}_1, x =^{\kappa} \underline{\tilde{e}} \equiv \underline{\tilde{d}}_0 \end{array}$$

Figure B.4: Definition of $\texttt{infer}(\Delta_0, \underline{\tilde{d}}, \Delta_1$

$$\begin{array}{l}\texttt{infer}(\Delta, x =^{\kappa_0} \underline{\tilde{e}}, x : \rho^{[\eta,\xi]}_{\kappa_1}) = \quad (\Pi, \Gamma) \\ \qquad\qquad \text{where } (\Pi_0, \Gamma) = \texttt{infer}(\Delta, \underline{\tilde{e}}, \rho^{[\eta \oplus \kappa_0, \xi \oplus \kappa_0]}) \\ \qquad\qquad\quad \Pi = \Pi_0, \kappa_1 \leq \kappa_0 \end{array}$$

Figure B.5: Definition of $\texttt{infer}(\Delta_0, \underline{\tilde{b}}, x : \sigma)$

Appendix C

An example program

This appendix contains an example of the analysis in action on the smallest program which we used to measure the efficiency of the analysis (see chapter 6). The program checks whether a string is a substring of another string and the original Haskell source code was as follows.

```
substring xs ys = any (isPrefixOf xs) (tails ys)
```

Since our small language does not contain characters we have used the function to check whether the list of integers [10..20] is a sublist of [1..]. The result of desugaring the program (and the predefined functions it uses) and analysing the program is given below.

```
let substring =✓ λ[0,0]xs.λ[0,0]ys.let p =! isPrefixOf xs
                                       ts =✓ tails ys
                                   in any p ts
    any =✓ λ[0,0]p.λ[0,0]xs.let bs =✓ map p xs
                            in or bs
    map =! λ[1,1]f.λ[0,0]xs.case xs of
                              nil ⇒ nil[0,0]
                              cons y ys ⇒ let z =✓ f y
                                             zs =✓ map f ys
                                          in cons[0,0] z zs
    or =✓ λ[0,0]bs.foldr orbin false bs
    foldr =! λ[1,1]f.λ[0,0]z.λ[0,0]xs.case xs of
                                    nil ⇒ z
                                    cons y ys ⇒ let r =✓ foldr f z ys
                                                in f y r
    orbin =! λ[1,1]a.λ[0,0]b.if a
```

```
                                then true[0,0]
                                else b
    false =✓ false[0,0]
    isPrefixOf =! λ[1,1]xs.λ[0,1]ys.
                    case xs of
                      nil ⇒ true[0,0]
                      cons z zs ⇒ case ys of
                                    nil ⇒ false[0,0]
                                    cons w ws ⇒ if z ==[0,0] w
                                                  then isPrefixOf zs ws
                                                  else false[0,0]
    tails =! λ[1,1]xs.case xs of
                        nil ⇒ let empty1 =✓ nil[0,0]
                                  empty2 =✓ nil[0,0]
                              in cons[0,0] empty1 empty2
                        cons y ys ⇒ let yss =✓ tails ys
                                    in cons[0,0] xs yss
    from =! λ[1,1]n.let n2 =! n +[1,1] 1[0,0]
                        ns =! from n2
                    in cons[1,1] n ns
    fromto =! λ[1,1]n.λ[0,0]m.if n >[0,0] m
                                then nil[1,1]
                                else let n2 =! n +[1,1] 1[0,0]
                                         ns =! fromto n2 m
                                     in cons[1,1] n ns
    one =! 1[1,1]
    ten =! 10[1,1]
    twenty =! 20[1,1]
    ns =! fromto ten twenty
    ms =! from one
in substring ns ms
```

Bibliography

[Abr90] Samson Abramsky. Computational interpretations of linear logic. Technical Report DOC 90/20, Imperial College, Department of Computing, 1990.

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[Arg89] Guy Argo. Improving the Three Instruction Machine. In *Proceedings 1989 conference on Functional Programming Languages and Computer Architecture*, 1989.

[BHY88] A. Bloss, P. Hudak, and J. Young. Code optimisations for lazy evaluation. *Lisp and Symbolic Computation*, 1:167–164, September 1988.

[BRJ88] G. Burn, J. Robson, and S. Peyton Jones. The Spineless G-machine. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, Snowbird, Utah, 1988.

[BS96] E. Barendsen and S. Smetsers. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings 1982 Symposium on Principles of Programming Languages*, Alberque, N.M, 1982.

[Fai85] Jon Fairbairn. Removing Redundant Laziness from Supercombinators. In *Workshop on Implementation of Functional Languages*, pages 181–189. Programming Methodology Group Chalmers University of Technology. PMG Report 17, 1985.

[Fax95] Karl-Filip Faxén. Optimizing lazy functional programs using flow inference. In *Second International Symposium on Static Analysis*, pages 136–153. Springer-Verlag, LNCS 983, 1995.

[Fax96] Karl-Filip Faxén. Flow Inference, Code Generation and Garbage Collection for Lazy Functional Languages. 1996.

[FW87] Jon Fairbairn and Stuart Wray. TIM: A Simple, Lazy Abstract Machine to Execute Supercombinators. In *IFIP conference on Functional Programming Languages and Computer Architecture*, pages 34–45. Springer Verlag LNCS 274, 1987.

[GH90] J. Guzmàn and P. Hudak. Single-Threaded Polymorphic Lambda Calculus. In *Symposium on Logic in Computer Science*, pages 333–343, 1990.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GL87] J. Y. Girard and Y. Lafont. Linear Logic and Lazy Computation. In *Proceedings of TAPSOFT '87, Volume 2*, pages 52–66. Springer-Verlag, LNCS 250, 1987.

[Gol87] Benjamin Goldberg. Detecting Sharing of Partial Applications in Functional Programs. In *Functional Programming Languages and Computer Architecture*, pages 408–425. Springer-Verlag, LNCS 274, 1987.

[GS91] Carsten K. Gomard and Peter Sestoft. Evaluation Order Analysis for Lazy Data Structures. In *Proc. 1991 Glasgow Workshop on Functional Programming*, Workshops in Computing. Springer–Verlag, 1991.

[Gus98] Jörgen Gustavsson. A Type Based Sharing Analysis for Update Avoidance and Optimisation. In *ACM SIGPLAN International Conference on Functional programming*, pages 39–50, Baltimore, Maryland, 1998.

[Hol88] Sören Holmström. A Linear Functional Language. In *Proceedings of the 1988 Workshop on Implementation of Lazy Functional Languages*, 1988.

[Hug88] J. Hughes. Backwards Analysis of Functional Programs. In Bjørner and Ershov, editors, *IFIP Workshop on Partial Evaluation and Mixed Computation*, pages 187–208, 1988.

[JHH+93] S.L. Peyton Jones, C.V. Hall, K. Hammond, W.D. Partain, and P.L. Wadler. The Glasgow Haskell compiler: a technical overview. In *Joint Framework for Information Technology (JFIT), Technical Conference Digest*, 1993.

[JP93] Simon Peyton Jones and Will Partain. Measuring the effectiveness of a simple strictness analyser. In *Proceedings of the Glasgow workshop on functional programming*, Workshops in Computing, pages 201–220. Springer–Verlag, 1993.

[Kob99] Naoki Kobayashi. Quasi-Linear Types. In *Proceedings 1999 Symposium on Principles of Programming Languages*, 1999.

[Laf88] Yves Lafont. The Linear Abstract Machine. *Theoretical Computer Science*, 59:157–180, 1988.

[Lau93] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proceedings 1993 Symposium on Principles of Programming Languages*, Charleston, N. Carolina, 1993.

[LGH+92] J. Launchbury, A. Gill, J. Hughes, S. Marlow, S. L. Peyton Jones, and P. Wadler. Avoiding Unnecessary Updates. In J. Launchbury and P. M. Sansom, editors, *Functional Programming*, Workshops in Computing, Glasgow, 1992.

[Mac94] Ian Mackie. Lilac: a functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):395–433, October 1994.

[Mar93] S. Marlow. Update Avoidance Analysis by Abstract Interpretation. In *Proc. 1993 Glasgow Workshop on Functional Programming*, Workshops in Computing. Springer–Verlag, 1993.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.

[Mog97] T. Mogensen. Types for 0, 1 or many uses. In *Proceedings of IFL '97: 9th International Workshop on Implementation of Functional Languages*, pages 112–122, St. Andrews, Scotland, September 1997. Springer-Verlag, LNCS 1467.

[Mos97] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs (Revised Version)*. PhD thesis, DIKU, University of Copenhagen, Denmark, August 1997.

[MS99] Andrew Moran and David Sands. Improvement in a Lazy Context: An Operational Theory for Call-By-Need. In *Proceedings 1999 Symposium on Principles of Programming Languages*, 1999.

[Myc82] Alan Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs*. PhD thesis, University of Edinburg, 1982.

[NNH98] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. November 1998. Draft distriduted at the advanced course on Principles of Program Analysis at Schloss Dagstuhl, Germany.

[PJ92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, July 1992.

[RW93] Colin Runciman and David Wakeling. Heap Profiling of Lazy Functional Programs. *Journal of Functional Programming*, 3(2):217–245, April 1993.

[Ses91] P. Sestoft. *Analysis and Efficient Implementation of Functional Programs*. PhD thesis, DIKU, University of Copenhagen, Denmark, October 1991.

[Ses97] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[TWM95] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *ACM Conf. on Functional Programming Languages and Computer Architecture*, La Jolla, 1995.

[Wad90] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland.

[Wad91] Philip Wadler. Is there a use for linear logic? In *ACM Conference on Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, Connecticut, June 1991.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[WJ98] Keith Wansbrough and Simon Peyton Jones. Once Upon a Polymorphic Type. Technical Report TR-1998-19, Department of Computing Science, University of Glasgow, 1998.

[WJ99a] Keith Wansbrough and Simon Peyton Jones. Personal communication, 1999.

[WJ99b] Keith Wansbrough and Simon Peyton Jones. Once Upon a Polymorphic Type. In *Proceedings 1999 Symposium on Principles of Programming Languages*, 1999.

[WR91] D. Wakeling and C. Runciman. Linearity and Laziness. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 215–240. Springer-Verlag, August 1991. LNCS 523.

[Wri96] David A. Wright. Linear, Strictness and Usage Logics. In *Proceedings of Computing: The Australasian Theory Symposium (CATS '96)*, Melbourne, Australia, 1996.

Paper IV

A Usage Analysis with Bounded Usage Polymorphism and Subtyping

A Usage Analysis with Bounded Usage Polymorphism and Subtyping

Jörgen Gustavsson Josef Svenningsson

Abstract

Usage analysis aims to predict the number of times a heap allocated closure is used. Previously proposed usage analyses have proved not to scale up well to large programs. In this paper we present a powerful and accurate type based analysis designed to scale up for large programs. The key features of the type system are usage subtyping and bounded usage polymorphism. Bounded polymorphism can lead to huge constraint sets so to express constraints compactly we introduce a new expressive form of constraints which allows constraints to be represented compactly through calls to *constraint abstractions*.

1 Introduction

In the implementation of a lazy functional language sharing of evaluation is performed by updating. For example, the (unoptimised) evaluation of

$$(\lambda x.x + x)\ (1 + 2)$$

proceeds as follows. First, a closure for $1 + 2$ is built in the heap and a reference to the closure is passed to the abstraction. Second, to evaluate $x + x$ the value of x is required. Thus the closure is fetched from the heap and evaluated. Third, the closure is updated with the result so that when the value of x is required again the expression needs not be recomputed.

Measurements by Marlow show that 70% of all closures are used at most once and that it is therefore unnecessary to update them. Usage information also enables a series of program transformations such as more aggressive inlining and let-floating [TWM95, WPJ99, GS99]. It is therefore no surprise that considerable effort has been put into static analyses that can discover if a closure is used at most once [Ses91, LGH+92, Mar93, TWM95, Fax95, BJ96, Mog97, Gus98, WPJ99]. This line of research has produced analyses with increasing accuracy, and benchmarks have shown that for small programs they discover a large portion of closures used at most once. However these analyses are monovariant and do not take the context where a function is called into account. When analysing large programs it is crucial to take the context into

account – when Wansbrough and Peyton Jones implemented the recent analysis from [WPJ99] into the Glasgow Haskell Compiler they discovered that it was almost useless in practice since it did not scale up for large programs. [WPJ00].

In this paper we present a powerful and accurate type system which attempts to solve this problem. It takes the context where a function is called into account through bounded usage polymorphism. We designed our type system by putting together and extending the best ideas from previous work. The salient features of the type system are these:

- Our system has full-blown bounded usage polymorphism and supports usage polymorhic recursion.
- In [WPJ98] Wansbrough and Peyton Jones give an overview of the design space for how to treat data structures. We choose the most aggressive approach which corresponds to the hard-wired treatment of lists in [TWM95].
- Our system is based on subsumption between usage types. The use of subtyping in usage analysis goes back to Faxén [Fax95].
- We have a three-level type language which incorporates separate notions of usage of closures and usage of values which gives increased precision. To separate the usage of closures and values is an idea due to Faxén [Fax95].
- We have expressive update annotations which allow us to express more aggressive optimisations than previous analyses.

Having all these features is not very useful unless there is an efficient inference algorithm for the type system. Here bounded polymorphism presents a problem. See for example Mossin's thesis [Mos97] for an account of the problems with bounded flow polymorphism in type based flow analyses. The core of the problem is that the quantified variables in a type schema may be constrained by a huge number of constraints. In the naive inference algorithm first presented by Mossin the number of constraints may be exponential in the size of the program. Mossin refines the algorithm by adding a constraint simplification phase which renders an inference algorithm which is $O(n^7)$.

A novelty in our work is a new expressive form of constraints which allows constraints to be represented compactly through calls to *constraint abstractions*. To efficiently compute least solutions to constraints with constraint abstractions is an involved problem and is the subject of a companion paper [GS01]. There we show how to efficiently compute a least solution to constraints in a constraint language with constraint abstractions and inequality constraints over a lattice. Using these techniques we can obtain an inference algorithm for our usage analysis which is $O(n^3)$ where n is the size of the explicitly typed program. We believe that constraint abstractions can be very useful for a range of program analyses which features bounded annotation polymorphism and in [GS01]

we show how to apply the ideas to a flow analysis with bounded flow polymorphism. Other candidates may be effect analysis, e.g., [TJ94], binding time analysis, e.g., [DHM95], non determinism analysis, e.g., [PS00] and uniqueness type systems, e.g., [BS96].

1.1 Outline

This paper is organised as follows. Section 2 introduces the language and its semantics. Section 3 presents the type system. Section 4 describes related work. Section 5 concludes.

2 Language

In this section we will present our language and its semantics in the form of an abstract machine.

2.1 Syntax

The language we use is a lambda calculus extended with integers, lists, case-expressions and recursive let-expressions. We omit user defined data structures to simplify the presentation but it is a straightforward matter to add them [Sve00].

Variables	x, y, z		
Values	v	$::=$	$\lambda x.e \mid n \mid \texttt{nil} \mid \texttt{cons}\ x\ y$
Expressions	e	$::=$	$v^{\kappa} \mid x \mid e\ x \mid e_0 +^{\kappa} e_1 \mid \texttt{let}\ b_1, \ldots, b_n\ \texttt{in}\ e \mid \texttt{case}\ e\ \texttt{of}\ alts$
Bindings	b	$::=$	$x =^{\kappa} e$
Alternatives	$alts$	$::=$	$\{\texttt{nil} \Rightarrow e_0; \texttt{cons}\ x\ y \Rightarrow e_1\}$
Annotations	κ	$::=$	$1 \mid \omega$

We annotate bindings, values and $+$ with usage annotations 1 and ω ranged over by κ. The intuitive meaning of 1 and ω is that the annotated binding (or value) may be used at most once and any number of times respectively.

A distinguishing feature of the syntax is that arguments (in applications of terms and constructors) are restricted to variables. We will occasionally use unrestricted application $e_0\ e_1$ as syntactic sugar for $\texttt{let}\ x =^{\omega} e_1\ \texttt{in}\ e_0\ x$ where x is a fresh variable. The purpose of the restricted syntax is to make the creation of closures explicit via a let-expression which greatly simplifies the presentation of the abstract machine as well as the analysis presented in this paper. The syntactic restriction is by now rather standard, see for example [PJPS96, Lau93, Ses97, GS99].

2.2 Semantics

We will take Sestoft's abstract machine [Ses97] as the semantic basis of our work. The machine can be thought of as modelling lower-level abstract machines based on so called update markers, such as the TIM [FW87] and the STG-machine [PJ92]. A correspondence between Sestoft's machine and Launchbury's natural semantics for lazy evaluation [Lau93] has been shown in [Ses97]. For the purpose of the abstract machine we extend the set of terms to include expressions of the form $\mathtt{add}_n^\kappa\ e$, which represents an intermediate step in the computation of $n^{\kappa'} +^\kappa e$. We define a reduction relation $e \mapsto e'$ between terms:

$$(\lambda x.e)^\kappa\ y \mapsto e[x{:=}y] \qquad n^{\kappa'} +^\kappa e \mapsto \mathtt{add}_n^\kappa\ e \qquad \mathtt{add}_{n_0}^\kappa\ n_1^{\kappa'} \mapsto \lceil n_0 + n_1 \rceil^\kappa$$

$$\left(\begin{array}{l}\mathtt{case}\ \mathtt{nil}^\kappa\ \mathtt{of}\\ \quad \mathtt{nil} \Rightarrow e_0\\ \quad \mathtt{cons}\ x'\ y' \Rightarrow e_1\end{array}\right) \mapsto e_0 \qquad \left(\begin{array}{l}\mathtt{case}\ (\mathtt{cons}\ x\ y)^\kappa\ \mathtt{of}\\ \quad \mathtt{nil} \Rightarrow e_0\\ \quad \mathtt{cons}\ x'\ y' \Rightarrow e_1\end{array}\right) \mapsto e_1[x'{:=}x, y'{:=}y]$$

Note that no reduction depends on an annotation. The annotations are instead taken into account in the abstract machine transition rules.

Configurations in the abstract machine are triples $\langle H\ ;\ e\ ;\ S\rangle$, where H is a heap, e is the term currently being evaluated and S is the abstract machine stack:

Heaps	H	$::= \quad b_1, \ldots, b_n$
Stacks	S	$::= \quad \epsilon \mid R, S \mid \#x, S$
Reduction contexts	R	$::= \quad [\cdot]\ x \mid [\cdot] +^\kappa e \mid \mathtt{add}_n^\kappa\ [\cdot] \mid \mathtt{case}\ [\cdot]\ \mathtt{of}\ alts$

A heap consists of a sequence of bindings. The variables bound by the heap must be distinct and the order of bindings is irrelevant. Thus a heap can be considered as a partial function mapping variables to terms and we will write $\mathrm{dom}(H)$ for the set of variables bound by H. We will write H_0, H_1 for the concatenation of H_0 and H_1. An abstract machine stack is a stack of shallow reduction contexts and update markers. The stack can be thought of as corresponding to the "surrounding derivation" in a natural semantics, where the rôle of an update marker $\#x$ is to keep track of a pending update of x. The update markers on the stack will be distinct, that is there will be no more than one pending update of the same variable. We will consider an update marker as a binder and we will write $\mathrm{dom}(S)$ for the variables bound by the update markers in S. Consequently, we will require the variables bound by the stack to be distinct from the variables bound by the heap. We will also require that configurations are closed and we will identify configurations up to α-conversion, that is renaming of the variables bound by the heap and the stack. We will also identify configurations up to garbage meaning that we may remove or add bindings and update markers to the heap as long as the configuration remains closed. An initial configuration is of the form $\langle \epsilon\ ;\ e\ ;\ \epsilon\rangle$, where e is a closed expression. The transition rules of the abstract machine are given in Figure 1. The rule Let

$$
\begin{array}{lcll}
\langle H\ ;\ \mathtt{let}\,\vec{b}\,\mathtt{in}\,e\ ;\ S\rangle & \overset{\text{Let}}{\longmapsto} & \langle H, \vec{b}\ ;\ e\ ;\ S\rangle & \\
\langle H, x =^{\omega} e\ ;\ x\ ;\ S\rangle & \overset{\text{Var-}\omega}{\longmapsto} & \langle H\ ;\ e\ ;\ \#x, S\rangle & \\
\langle H, x =^{1} e\ ;\ x\ ;\ S\rangle & \overset{\text{Var-1}}{\longmapsto} & \langle H\ ;\ e\ ;\ S\rangle & \\
\langle H\ ;\ R[e]\ ;\ S\rangle & \overset{\text{Unwind}}{\longmapsto} & \langle H\ ;\ e\ ;\ R, S\rangle & \\
\langle H\ ;\ v^{\kappa}\ ;\ R, S\rangle & \overset{\text{Reduce}}{\longmapsto} & \langle H\ ;\ e\ ;\ S\rangle & \text{if } R[v^{\kappa}] \mapsto e \\
\langle H\ ;\ v^{\omega}\ ;\ \#x, S\rangle & \overset{\text{Marker-}\omega}{\longmapsto} & \langle H, x =^{\omega} v^{\omega}\ ;\ v^{\omega}\ ;\ S\rangle & \\
\langle H\ ;\ v^{1}\ ;\ \#x, S\rangle & \overset{\text{Marker-1}}{\longmapsto} & \langle H\ ;\ v^{1}\ ;\ S\rangle &
\end{array}
$$

Figure 1: Abstract machine transition rules

$$\langle H\ ;\ \mathtt{let}\,\vec{b}\,\mathtt{in}\,e\ ;\ S\rangle \overset{\text{Let}}{\longmapsto} \langle H, \vec{b}\ ;\ e\ ;\ S\rangle$$

creates new bindings in the heap. For the rule to be applied the variables bound by $\vec{b}$ must be distinct from the variables bound by H and S. This condition can always be met simply by α-converting the let-expression. The rule Var-ω

$$\langle H, x =^{\omega} e\ ;\ x\ ;\ S\rangle \overset{\text{Var-}\omega}{\longmapsto} \langle H\ ;\ e\ ;\ \#x, S\rangle$$

gives semantics to bindings annotated with ω. The rule states that an update marker shall be pushed onto the stack so that the variable x eventually may be updated with the result of evaluating e. The removal of the binding corresponds to so called black-holing: if the evaluation of e to a value depends on x (i.e., x depends directly on itself) the computation will get stuck, since x is no longer bound by the heap. Note that we still consider the configuration to be closed, since x is bound by the update marker on the stack. The rule Var-1

$$\langle H, x =^{1} e\ ;\ x\ ;\ S\rangle \overset{\text{Var-1}}{\longmapsto} \langle H\ ;\ e\ ;\ S\rangle$$

gives semantics to bindings annotated with 1. Such bindings may only be used once so there is no need to update the binding and thus no update marker is pushed onto the stack. Note that we require configurations to be closed so the rule does not apply unless the configuration remains closed. An example of where the rule does not apply is the configuration

$$\langle x =^{1} 1 +^{\omega} 2\ ;\ x\ ;\ [\cdot] +^{\kappa} x, \epsilon\rangle$$

which cannot reduce further since there is a reference to x on the stack. This restriction is important since an open configuration would correspond to dangling pointers in an implementation. If the rule does not apply the computation will go *wrong*, and we will consider the configuration and the term it originates from to be ill-annotated. The key property of the type system presented in

this paper is that if a term is well-typed then it cannot go wrong. Note that, the insistence that configurations remain closed is a stronger requirement than the intuitive "used at most once" criterion, which says that it is safe to avoid updating a closure if it is used at most once. For example, according to the weaker criterion it is safe to not update x in

$$\texttt{let}\, x = 1 + 2\, \texttt{in}\, x + (\lambda y.3)\, x$$

because x is only used once, but according to our criterion it is not safe. Our stronger criterion is useful for two reasons. Firstly, with dangling pointers special care has to be taken so that the garbage collector does not follow them – and there is a cost associated with that. Secondly, usage annotations can be used to justify certain program transformations, such as more aggressive inlining. Gustavsson and Sands [GS99] have shown that the stronger criterion can guarantee that these transformations are time and space safe, but with the weaker "used at most once" criterion the transformations can lead to an asymptoticly worse space behaviour. The rule Unwind

$$\langle H\ ;\ R[e]\ ;\ S\rangle \overset{\text{Unwind}}{\longmapsto} \langle H\ ;\ e\ ;\ R, S\rangle$$

allows us to get to the heart of the evaluation by "unwinding" a shallow reduction context. When the term to be evaluated is a value the next transition depends on whether an update marker or a reduction context is on top of the stack. If it is a reduction context the rule Reduce

$$\langle H\ ;\ v\ ;\ R, S\rangle \overset{\text{Reduce}}{\longmapsto} \langle H\ ;\ e\ ;\ S\rangle \quad \text{if } R[v] \mapsto e$$

applies, the value is plugged into the reduction context and a reduction can take place. If the top of the stack is an update marker, what happens depends on the annotation on the value. If it is ω the value may be used several times and we apply the rule Update-ω

$$\langle H\ ;\ v^{\omega}\ ;\ \#x, S\rangle \overset{\text{Marker-}\omega}{\longmapsto} \langle H, x =^{\omega} v^{\omega}\ ;\ v^{\omega}\ ;\ S\rangle$$

which takes care of the update marker and performs the update. If the value on the other hand is annotated with 1, the value may only be used once so the rule Update-1

$$\langle H\ ;\ v^{1}\ ;\ \#x, S\rangle \overset{\text{Marker-1}}{\longmapsto} \langle H\ ;\ v^{1}\ ;\ S\rangle$$

throws away the marker without performing the update. Again, note that the rule does not apply unless the configuration remains closed. So, for example,

$$\langle \epsilon\ ;\ 3^{1}\ ;\ \#x, [\cdot] +^{\kappa} x, \epsilon\rangle$$

goes wrong and we consider the configuration to be ill-annotated.

3 Type system

The semantics in Section 2 specifies that for a binding $x = e$ to be safely annotated with a 1 it is required that whenever the binding is used through the rule

$$\langle H, x =^1 e\ ;\ x\ ;\ S\rangle \overset{\text{Var-1}}{\longmapsto} \langle H\ ;\ e\ ;\ S\rangle,$$

the configuration must remain closed. Thus there may only be one (non-binding) occurrence of x in the configuration, namely the one that is dereferenced. Similarly, to safely annotate a value with 1 it is required that if and when the value is used and there is an update marker $\#x$ on the stack

$$\langle H\ ;\ v^1\ ;\ \#x, S\rangle \overset{\text{Marker-1}}{\longmapsto} \langle H\ ;\ v^1\ ;\ S\rangle \quad ,$$

then there is no live occurrence of x in the configuration so that the configuration remains closed. Our type system (and most other type based usage analyses) is based on the following simple idea. If, when a binding $x = e$ is created, x occurs only once in the configuration and x never gets duplicated during the computation then x will occur only once if and when it is dereferenced. [1]

3.1 Type language

In order to construct a type system for the annotated language we need a corresponding annotated type language. We start by extending the annotation language from the previous section to include annotation variables.

$$\text{Annotations} \quad \kappa \quad ::= \quad 1 \mid \omega \mid k \mid j$$

We will use two kinds of variables, *type annotation variables*, ranged over by k, and *program annotation variables*, ranged over by j. Type annotation variables may occur in the annotations on a type but not in the annotations on a program. Conversely, program annotation variables may occur in programs but not in types.

The structure of the type language closely follows the structure of the term language and we will have one kind of type for every syntactic category. We let ρ range over *value types* which is the form of type we will assign to values.

$$\begin{array}{lll} \text{Type Variables} & a & \\ \text{Value Types} & \rho & ::= \quad a \mid \texttt{Int} \mid \sigma \to \tau \mid \texttt{List}\,\kappa_0\,\kappa_1\,\kappa_2\,\kappa_3\,\rho \end{array}$$

Our value types contains type variables, an integer type, function types and the list type. The function types relies on a notion of *binding types*, ranged over

[1] We will strengthen this idea in an obvious but important way – when a variable occurs once in several branches of a case-expression. Then, since eventually only one branch will be taken, we may consider it as occurring only once.

by σ, and *expression types*, ranged over by τ, which we will introduce below. Expression types are used to give types to expressions and are defined as follows.

$$\text{Expression Types} \quad \tau \quad ::= \quad \rho^{\kappa}$$

An annotated value v^{κ} will be given a type of the form ρ^{κ} and a non-value e will be given a type such that the annotated value of e (if e terminates) will have that type. Thus, for example, saying that a term has a type ρ^{ω} means that the value of the term may be used any number of times. Binding types which we will use to give a type to bindings are defined as follows.

$$\text{Binding Types} \quad \sigma \quad ::= \quad \tau_{\kappa}$$

A binding $x =^{\kappa} e$ may be given a type of the form τ_{κ} where τ is the type of e. We also use binding types to give a type to a variable when we can think of the variable as a reference, for example when we pass it as an argument to a function. A type of a variable is then simply the type of the bindings it may refer to. Recall that we used expression types and binding types in the type $\sigma \to \tau$ of a function. A function of this type can be applied to a variable (remember functions can only be applied to variables due to the syntactic restriction in our language) with the binding type σ and then it will return something of type τ. We can also use binding types to logically justify our type $\mathtt{List}\,\kappa_0\,\kappa_1\,\kappa_2\,\kappa_3\,\rho$ of lists. We can obtain this type simply by annotating the right hand side of the data type definition

$$\mathtt{List}\,a = \mathtt{nil} \mid \mathtt{cons}\,a\,(\mathtt{List}\,a)$$

such that the arguments to the constructors are binding types, as follows.

$$\mathtt{List}\,k_0\,k_1\,k_2\,k_3\,a = \mathtt{nil} \mid \mathtt{cons}\,a^{k_1}_{k_0}\,(\mathtt{List}\,k_0\,k_1\,k_2\,k_3\,a)^{k_3}_{k_2}$$

The reason for why the arguments to the constructors should be binding types is simply because constructors, due to the syntactic restriction, may be applied only to variables.

3.2 Subtyping

A key observation which we will use to justify our subtyping relation is that 1 operationally approximates ω, i.e., if we in any term e replace any occurrence of 1 with ω then the modified term will run successfully without going wrong if and when e does. We define the subtyping relation on *closed* types where the ordering on annotations is the operational approximation $1 < \omega$ by the following rules.

$$\frac{\sigma' \leq \sigma \qquad \tau \leq \tau'}{\sigma \to \tau \leq \sigma' \to \tau'} \qquad \frac{\rho^{\kappa_1}_{\kappa_0} \leq \rho'^{\kappa_1'}_{\kappa_0'} \qquad \kappa_2' \leq \kappa_2 \qquad \kappa_3' \leq \kappa_3}{\mathtt{List}\,\kappa_0\,\kappa_1\,\kappa_2\,\kappa_3\,\rho \leq \mathtt{List}\,\kappa_0'\,\kappa_1'\,\kappa_2'\,\kappa_3'\,\rho'}$$

$$\frac{}{\mathtt{Int} \leq \mathtt{Int}} \qquad \frac{\rho \leq \rho' \qquad \kappa' \leq \kappa}{\rho^{\kappa} \leq \rho'^{\kappa'}} \qquad \frac{\tau \leq \tau' \qquad \kappa' \leq \kappa}{\tau_{\kappa} \leq \tau'_{\kappa'}}$$

Note that the subtype ordering is contravariant with respect to the ordering on the annotations. The rule for lists can be understood by unfolding the annotated data type definition for lists.

3.3 Constraints

In order to extend the subtyping relation to types with type variables and annotation variables we need the notion of constraints. To be able to represents constraints compactly we introduce a new form of constraints which may contain calls to *constraint abstractions*. A constraint abstraction is simply a function that given some annotation variables returns a constraint. We will let ϕ range over constraint abstractions, l range over constraint abstraction variables and Π range over constraints.

$$\begin{array}{lrcl} \text{Annotation constraints} & \Pi & ::= & \kappa_0 \leq \kappa_1 \mid \Pi_0, \Pi_1 \mid \text{let } \vec{\phi} \text{ in } \Pi \mid \exists \vec{k}.\Pi \mid l\ \vec{\kappa} \\ \text{Constraint abstractions} & \phi & ::= & l\ \vec{k} = \Pi \end{array}$$

Constraint abstractions allow different substitution instances of a constraint to share the same representation. For example to represent instances of the constraints $k_0 \leq k_1,\ k_1 \leq k_2$ we can define an abstraction

$$l\ k_0\ k_1\ k_2 = k_0 \leq k_1,\ k_1 \leq k_2$$

and represent $(\kappa_0 \leq \kappa_1,\ \kappa_1 \leq \kappa_2), (\kappa_3 \leq \kappa_4,\ \kappa_4 \leq \kappa_5)$ as

$$\text{let } l\ k_0\ k_1\ k_2 = k_0 \leq k_1,\ k_1 \leq k_2 \text{ in } l\ \kappa_0\ \kappa_1\ \kappa_2,\ l\ \kappa_3\ \kappa_4\ \kappa_5.$$

Thus with constraint abstractions the size of any instance is linear in the number of free type annotation variables of the constraint but the size of the original constraint may be quadratic in the sum of the number of free type annotation variables and free program annotation variables (or even worse if it contains existential quantifiers). With constraint abstraction we can avoid the exponential explosion of constraints which can happen with a naive approach. To see why consider a program of the following form.

$$\begin{array}{l} \texttt{let } f_0 = \ldots \\ \texttt{in let } f_1 = \ldots f_0 \ldots f_0 \ldots \\ \quad \texttt{in let } \ldots \\ \quad\quad \texttt{in let } f_n = \ldots f_{n-1} \ldots f_{n-1} \ldots \\ \quad\quad\quad \texttt{in } \ldots f_n \ldots f_n \ldots \end{array}$$

The first naive algorithm, for the similar problem of flow analysis with bounded flow polymorphism, presented by Mossin [Mos97] which suffers from the exponential explosion problem would proceed as follows. It first infers the polymorphic type for f_0. Then to compute the type for f_1 it instantiates the type of f_0 twice and thus make two instances of the constraints contained in the type

schema so the constraints for f_1 will be at least twice as big. This is repeated n times and thus the size of the resulting constraints will be exponential in the call depth n. In practice the call depth typically does not grow linearly with the size of the program but the call depth does tend to increase with program size which makes this into a problem that occurs in practice. With constraint abstractions we can avoid the problem and represent the constraints as follows

$$\begin{array}{l}\text{let } l_0\ \vec{k}_0 = \ldots \\ \text{in let } l_1\ \vec{k}_1 = \ldots l_0\ \vec{k}'_0 \ldots l_0\ \vec{k}''_0 \ldots \\ \quad \text{in let } \ldots \\ \quad\quad \text{in let } l_n\ \vec{k}_n = \ldots l_{n-1}\ \vec{k}'_{n-1} \ldots l_{n-1}\ \vec{k}''_{n-1} \ldots \\ \quad\quad\quad \text{in } \ldots l_n\ \vec{k}'_0 \ldots l_n\ \vec{k}''_0 \ldots\end{array}$$

To give semantics to constraints we will use closing substitutions from type variables to value types and annotation variables to annotations, ranged over by ϑ. The meaning of a constraint Π is given by a relation $\vartheta; \vec{\phi} \models \Pi$ (read as $\vartheta; \vec{\phi}$ models Π) defined coinductively by the following rules.

$$\frac{\kappa_0\vartheta \leq \kappa_1\vartheta}{\vartheta; \vec{\phi} \models \kappa_0 \leq \kappa_1} \qquad \frac{\vartheta; \vec{\phi} \models \Pi_0 \qquad \vartheta; \vec{\phi} \models \Pi_1}{\vartheta; \vec{\phi} \models \Pi_0, \Pi_1} \qquad \frac{\vartheta; \vec{\phi}, \vec{\phi}' \models \Pi}{\vartheta; \vec{\phi} \models \text{let } \vec{\phi}' \text{ in } \Pi}$$

$$\frac{\vartheta; \vec{\phi} \models \Pi[\vec{k} := \vec{\kappa}]}{\vartheta; \vec{\phi} \models \exists \vec{k}.\Pi} \qquad \frac{\vartheta; \vec{\phi} \models \Pi[\vec{k} := \vec{\kappa}]}{\vartheta; \vec{\phi} \models l\ \vec{\kappa}} \quad l\ \vec{k} = \Pi \in \vec{\phi}$$

We will sometimes write $\vartheta \models \Pi$ as a shorthand for $\vartheta; \epsilon \models \Pi$. We will let Ψ range over constraints concerning type variables.

$$\text{Type variable constraints} \quad \Psi ::= \quad a_0 \leq a_1 \mid \Psi_0, \Psi_1 \mid \exists \vec{a}.\Psi$$

The meaning of a constraint Ψ is given by a relation $\vartheta \models \Psi$ (read as ϑ models Ψ). We define $\vartheta \models \Psi$ inductively by the following rules.

$$\frac{\vartheta(a_0) \leq \vartheta(a_1)}{\vartheta \models a_0 \leq a_1} \qquad \frac{\vartheta \models \Psi_0 \qquad \vartheta \models \Psi_1}{\vartheta \models \Psi_0, \Psi_1} \qquad \frac{\vartheta[\vec{a} := \vec{\rho}] \models \Psi}{\vartheta \models \exists \vec{a}.\Psi}$$

We will let Θ range over pairs $\Pi; \Psi$ and we define $\vartheta \models \Theta$ as $\vartheta \models \Pi; \Psi$ iff $\vartheta \models \Pi$ and $\vartheta \models \Psi$. The whole purpose of having constraints is that they allow us to extend the subtyping relation to types with variables. We will define a relation $\Theta \models \rho_0 \leq \rho_1$ where ρ_0 and ρ_1 may be open types, which reads: $\rho_0 \leq \rho_1$ is a consequence of Θ. It is defined as $\Theta \models \rho_0 \leq \rho_1$ iff for every ϑ, if $\vartheta \models \Theta$ then $\rho_0\vartheta \leq \rho_1\vartheta$. We also define $\Theta \models \tau_0 \leq \tau_1$ and $\Theta \models \sigma_0 \leq \sigma_1$ in the same manner.

3.4 Type schemas

Our type system incorporates bounded polymorphism so we need type schemas where the quantified variables are bounded by some constraints.

$$\text{Type Schemas} \quad \chi ::= \quad \forall \vec{k}, \vec{a}.\ \rho \mid \Theta$$

We will define a relation $\Theta \models \chi \prec \rho$ which reads as: it is a consequence of Θ that χ can be instantiated to ρ. It is defined as $\Theta \models (\forall \vec{k}, \vec{a}.\, \rho \,|\, \Theta') \prec \rho[\vec{k} := \vec{\kappa}, \vec{a} := \vec{\rho}]$ iff for every ϑ, if $\vartheta \models \Theta$ then $\vartheta \circ [\vec{k} := \vec{\kappa}, \vec{a} := \vec{\rho}] \models \Theta'$. We will sometimes consider a value type ρ to be a type schema with no quantified variables and no constraints.

3.5 Contexts

We use Γ and Δ to range over typing contexts which are multisets of type associations of the form $x : \chi_{\kappa'}^{\kappa}$ (and since we may consider a value type ρ as a type schema there may also be type associations of the form $x : \rho_{\kappa'}^{\kappa}$). As usual we will use contexts when we give a type to a term with free variables. Thus we will say that e has the type τ in a context Γ if we can give e the type τ assuming that the free variables in e has the types given by Γ. However the context also plays another important rôle; it records the number of times each variable occurs in the term. Thus if x occurs n times in e it also occurs n times in Γ (with one important exception, namely if x occurs in different branches of a case-expression). This may be a bit surprising at first. Consider for example the term $(\lambda y.y +^1 y)^1\ x$ with the free variable x. We will be able to say that this term has the type $\mathtt{Int}^1$ in the context $x : \mathtt{Int}^\omega_\omega$. According to the reduction relation the term can reduce to $x +^1 x$ so we would expect to be able to give $x +^1 x$ the same type in the same context. However this will not be possible since x now occurs twice in the term. Instead we can type the term in the context $x : \mathtt{Int}^\omega_\omega, x : \mathtt{Int}^\omega_\omega$ where x occurs twice. To be able to state a relation between the contexts before and after a reduction we define a rewrite relation on contexts.

$$\Gamma,\ x : \chi^\omega_\omega \to \Gamma,\ x : \chi^\omega_\omega,\ x : \chi^\omega_\omega \qquad \Gamma,\ x : \chi^\kappa_{\kappa'} \to \Gamma$$

We have two rewrite rules. The first says that a type association of the form $x : \chi^\omega_\omega$ may be duplicated. This is supposed to model the duplication of a variable x during the computation. Note that we may not duplicate a type association of the form $x : \chi^1_1$. This reflects our intention that a variable that refers to a binding which will not be updated, must not be duplicated. The second rule simply allows us to remove a type association. This corresponds to the case when a variable is dropped during the computation (for example since it occurred in a branch of a case-expression that was not selected). These rewrite rules will play a rôle similar to the contraction and weakening rules in logic. The restricted duplication (i.e., that we may only duplicate type associations of the form $x : \chi^\omega_\omega$) corresponds to the restricted form of contraction in linear logic [Gir87]. We extend the relation to contexts with open types in the same way as with the subtyping relation by defining $\Theta \models \Gamma_0 \to^* \Gamma_1$ iff for every ϑ, if $\vartheta \models \Theta$ then $\Gamma_0\vartheta \to^* \Gamma_1\vartheta$. Finally we will also need the relation $\Theta \models \mathtt{if}\ \kappa = \omega\ \mathtt{then}\ \Gamma \to^* \Gamma, \Gamma$ which holds iff for every ϑ, if $\vartheta \models \Theta$, and $\kappa\vartheta = \omega$ then $\Gamma\vartheta \to^* \Gamma\vartheta, \Gamma\vartheta$.

$$\text{Abs}\frac{\Theta;\Gamma_0,\Gamma_1 \vdash e : \tau \quad x \notin \mathrm{dom}(\Gamma_0)}{\Theta;\Gamma_0 \vdash \lambda x.e : \sigma \to \tau \quad \Theta \models x : \sigma \to^* \Gamma_1}$$

$$\text{Int}\frac{}{\Theta;\emptyset \vdash n : \mathtt{Int}} \qquad \text{Nil}\frac{}{\Theta;\emptyset \vdash \mathtt{nil} : \mathtt{List}\,\kappa_0\,\kappa_1\,\kappa_2\,\kappa_3\,\rho}$$

$$\text{Cons}\frac{\rho' \equiv \mathtt{List}\,\kappa_4\,\kappa_5\,\kappa_6\,\kappa_7\,\rho \quad \Theta \models \chi_0 \prec \rho_0,\ \chi_1 \prec \rho_1 \quad \Theta \models {\rho_0}^{\kappa_1}_{\kappa_0} \leq \rho^{\kappa_5}_{\kappa_4},\ {\rho_1}^{\kappa_3}_{\kappa_2} \leq {\rho'}^{\kappa_7}_{\kappa_6}}{\Theta; x : \chi_0{}^{\kappa_1}_{\kappa_0}, y : \chi_1{}^{\kappa_3}_{\kappa_2} \vdash \mathtt{cons}\,x\,y : \rho'}$$

Figure 2: Typing rules for values

3.6 Typing judgements

Typing judgements for values take the form $\Theta;\Gamma \vdash v : \rho$ and shall be read: under the constraints Θ and in the context Γ, the value v can be given the value type ρ. Similarly we will have typing judgements for expressions, alternatives and bindings. As discussed in the previous section the context Γ in our judgements as usual keeps track of the types of the free variables in the term but it also records the number of times each variable occurs in the term.

3.7 Typing rules

The typing rules for values are in Figure 2. The key feature of the rule Abs

$$\frac{\Theta;\Gamma_0,\Gamma_1 \vdash e : \tau \quad x \notin \mathrm{dom}(\Gamma_0)}{\Theta;\Gamma_0 \vdash \lambda x.e : \sigma \to \tau \quad \Theta \models x : \sigma \to^* \Gamma_1}$$

is that if x occurs more than once in e then the abstraction will be assigned a type of the form $\rho^{\kappa}_{\kappa'} \to \tau$ where κ and κ' are constrained to be ω indicating that a variable will be duplicated if it is passed to the abstraction. This is accomplished by first typing e in a context Γ_0,Γ_1 where $x \notin \mathrm{dom}(\Gamma_0)$. Then, if x occurs more than once in e, x will occur more than once in Γ_1. Now the second side condition specify that we must be able to rewrite $x : \rho^{\kappa}_{\kappa'}$ to Γ_1 which clearly involves duplicating $x : \rho^{\kappa}_{\kappa'}$ (since x occurs more than once in Γ_1) which will constrain κ and κ' to be ω. The typing rule for integers is straightforward and the rules for lists can be understood by unfolding the annotated data type definition for lists.

We have divided the typing rules for expressions into two figures. Most rules appear in Figure 3 but the rules which concern let expressions are in Figure 4. The rule Value

$$\frac{\Theta;\Gamma \vdash v : \rho \quad \Theta \models \mathtt{if}\ \kappa' = \omega\ \mathtt{then}\ \Gamma \to^* \Gamma,\Gamma}{\Theta;\Gamma \vdash v^{\kappa} : \rho^{\kappa'} \quad \Theta \models \kappa' \leq \kappa}$$

$$\text{Value}\frac{\Theta;\Gamma \vdash v : \rho \quad \Theta \models \texttt{if}\ \kappa' = \omega\ \texttt{then}\ \Gamma \to^* \Gamma,\Gamma}{\Theta;\Gamma \vdash v^{\kappa} : \rho^{\kappa'} \quad \Theta \models \kappa' \leq \kappa}$$

$$\text{Var}\frac{}{\Theta; x : \chi^{\kappa_1}_{\kappa_0} \vdash x : \tau \quad \Theta \models \chi \prec \rho \quad \Theta \models \rho^{\kappa_1} \leq \tau} \qquad \text{App}\frac{\Theta;\Gamma \vdash e : (\rho^{\kappa_0}_{\kappa_1} \to \tau)^{\kappa}}{\Theta;\Gamma, x : \chi^{\kappa_0}_{\kappa_1} \vdash e\ x : \tau} \quad \Theta \models \chi \prec \rho$$

$$\text{Plus}\frac{\Theta;\Gamma_0 \vdash e_0 : \texttt{Int}^{\kappa_0} \qquad \Theta;\Gamma_1 \vdash e_1 : \texttt{Int}^{\kappa_1}}{\Theta;\Gamma_0,\Gamma_1 \vdash e_0 +^{\kappa} e_1 : \texttt{Int}^{\kappa'}} \quad \Theta \models \kappa' \leq \kappa$$

$$\text{Alts}\frac{\Theta;\Gamma_0,\Gamma_1 \vdash e_0 : \tau \qquad \Theta;\Gamma_0,\Gamma_2,\Gamma_3 \vdash e_1 : \tau}{\Theta;\Gamma_0,\Gamma_1,\Gamma_2 \vdash \{\texttt{nil} \Rightarrow e_0; \texttt{cons}\ x\ y \Rightarrow e_1\} : \rho' \Rightarrow \tau} \quad (*)$$

$$(*)\ \begin{array}{l} \rho' \equiv \texttt{List}\ \kappa_0\ \kappa_1\ \kappa_2\ \kappa_3\ \rho \\ x, y \notin \mathrm{dom}(\Gamma_0,\Gamma_2) \\ \Theta \models x : \rho^{\kappa_1}_{\kappa_0}, y : \rho'^{\kappa_3}_{\kappa_2} \to^* \Gamma_3 \end{array}$$

$$\text{Case}\frac{\Theta;\Gamma_0 \vdash e : \rho^{\kappa} \qquad \Theta;\Gamma_1 \vdash alts : \rho \Rightarrow \tau}{\Theta;\Gamma_0,\Gamma_1 \vdash \texttt{case}\ e\ \texttt{of}\ alts : \tau}$$

Figure 3: Typing rules for expressions

is used to type an annotated value. Saying that an annotated value has the type $\rho^{\kappa'}$ means that if κ' is ω the value may be used any number of times and thus it will take care of any update marker on the stack. Taking care of an update marker means updating with the value, thus duplicating any free variables of the value. The purpose of the side condition $\Theta \models \texttt{if}\ \kappa' = \omega\ \texttt{then}\ \Gamma \to^* \Gamma,\Gamma$ is to ensure that these variables may safely be duplicated if κ' is constrained to be ω.

In order to type case-expressions we introduce an auxiliary form of judgements for alternatives. We give alternatives a type of the form $\rho \Rightarrow \tau$ where ρ is the type of the value that is being scrutinised and τ is the type of the branches. The rule Alts

$$\frac{\Theta;\Gamma_0,\Gamma_1 \vdash e_0 : \tau \qquad \Theta;\Gamma_0,\Gamma_2,\Gamma_3 \vdash e_1 : \tau}{\Theta;\Gamma_0,\Gamma_1,\Gamma_2 \vdash \{\texttt{nil} \Rightarrow e_0; \texttt{cons}\ x\ y \Rightarrow e_1\} : \rho' \Rightarrow \tau} \quad \begin{array}{l} \rho' \equiv \texttt{List}\ \kappa_0\ \kappa_1\ \kappa_2\ \kappa_3\ \rho \\ x, y \notin \mathrm{dom}(\Gamma_0,\Gamma_2) \\ \Theta \models x : \rho^{\kappa_1}_{\kappa_0}, y : \rho'^{\kappa_3}_{\kappa_2} \to^* \Gamma_3 \end{array}$$

for alternatives contains a subtle treatment of contexts. If a variable occurs once in each branch of the case-expression and thus twice in the term it may still occur only once in the context. This is achieved by collecting the variables that occur in both branches in a common context Γ_0, thus effectively counting a variable occurring in both branches as one. Finally, the side conditions take care of the

$$\text{Binding}\ \frac{\Pi_0;\Psi;\Gamma\vdash e:\rho^{\kappa_0}}{\Pi;\Gamma\vdash x=^{\kappa}e:(x:(\forall\vec{k}_1,\vec{a}_1.\,\rho\mid l\vec{k}_2;\exists\vec{a}_0.\Psi)^{\kappa_0}_{\kappa_1})\ \mathbf{where}\ l\vec{k}_2=\exists\vec{k}_0.\Pi_0}\quad(*)$$

$$\text{Binding group-}\epsilon\ \frac{}{\Pi;\epsilon\vdash\epsilon:\epsilon\ \mathbf{where}\ \epsilon}\qquad(*)\ \begin{array}{l}\vec{k}_0\notin ftav(\Gamma,\rho^{\kappa}_0),\ \vec{a}_0\notin ftv(\Gamma,\rho^{\kappa}_0),\\ \vec{k}_1\notin ftav(\Gamma,\kappa_0,\ l\vec{k}_2=\exists\vec{k}_0.\Pi_0),\\ \vec{a}_1\notin ftv(\Gamma),\ \Pi\models\kappa_1\leq\kappa\end{array}$$

$$\text{Binding group}\ \frac{\Pi;\Gamma_0\vdash b:(x:\chi^{\kappa_1}_{\kappa_0})\ \mathbf{where}\ \phi\qquad\Pi;\Gamma_1\vdash\vec{b}:\Delta\ \mathbf{where}\ \vec{\phi}}{\Pi;\Gamma_0,\Gamma_1\vdash b,\vec{b}:(x:\chi^{\kappa_1}_{\kappa_0},\Delta)\ \mathbf{where}\ \phi,\vec{\phi}}$$

$$\text{Let}\ \frac{\Pi_0;\Gamma_0,\Gamma_1\vdash\vec{b}:\Delta\ \mathbf{where}\ \vec{\phi}\qquad\Pi_1;\Psi;\Gamma_2,\Gamma_3\vdash e:\tau}{\Pi;\Psi;\Gamma_1,\Gamma_3\vdash\mathtt{let}\,\vec{b}\,\mathtt{in}\,e:\tau}\quad(**)$$

$$(**)\ \begin{array}{l}\mathrm{dom}(\Gamma_1,\Gamma_3)\cap\mathrm{dom}(\Delta)=\emptyset\\ \Pi\models\Delta\rightarrow^{*}\Gamma_0;\Gamma_2\\ \Pi\models\Pi_0,\mathrm{let}\,\vec{\phi}\,\mathrm{in}\,\Pi_1\end{array}$$

Figure 4: Typing rules for bindings and let expressions

variables bound in the cons-pattern. They see to that if x (and/or y) occurs several times in e_1 then κ_0 and κ_1 (and/or κ_2 and κ_3) will be constrained to be ω. Thanks to the auxiliary rule for alternatives the rule for case-expressions becomes entirely straightforward.

To type let-expressions we first introduce an auxiliary form of typing judgements for bindings. We will give bindings a type of the form $x:\chi^{\kappa_0}_{\kappa_1}$, i.e., the type of a binding includes the name of the bound variable (so it can be considered as a type association). The rules for typing bindings appears in Figure 4. To type a binding with the rule Binding

$$\frac{\Pi_0;\Psi;\Gamma\vdash e:\rho^{\kappa_0}}{\Pi;\Gamma\vdash x=^{\kappa}e:(x:(\forall\vec{k}_1,\vec{a}_1.\,\rho\mid l\vec{k}_2;\exists\vec{a}_0.\Psi)^{\kappa_0}_{\kappa_1})\ \mathbf{where}\ l\vec{k}_2=\exists\vec{k}_0.\Pi_0}\quad(*)$$

$$(*)\ \begin{array}{l}\vec{k}_0\notin ftav(\Gamma,\rho^{\kappa}_0),\ \vec{a}_0\notin ftv(\Gamma,\rho^{\kappa}_0),\\ \vec{k}_1\notin ftav(\Gamma,\kappa_0,\ l\vec{k}_2=\exists\vec{k}_0.\Pi_0),\\ \vec{a}_1\notin ftv(\Gamma),\ \Pi\models\kappa_1\leq\kappa\end{array}$$

we first type the expression in the binding and yield the constraints $\Pi_0;\Psi$. We may then existentially quantify variables which appear in the constraints to obtain $\exists\vec{k}_0.\Pi_0$ and $\exists\vec{a}_0.\Psi$ providing $\vec{k}_0$ and $\vec{a}_0$ do not occur free elsewhere in the judgement. This is ensured by the first line of side conditions. We then form

the type schema $\forall \vec{k}_1, \vec{a}_1.\, \rho \,|\, l\vec{k}_2; \exists \vec{a}_0.\Psi$ by universally quantifying $\vec{k}_1$ and $\vec{a}_1$. The second line of side conditions simply ensures that $\vec{k}_1$ and $\vec{a}_1$ do not occur free elsewhere in the judgement. We put $\exists \vec{a}_0.\Psi$ in the type schema but not $\exists \vec{k}_0.\Pi_0$. Instead we introduce a constraint abstraction $l\vec{k}_2 = \exists \vec{k}_0.\Pi_0$ and put a call to the constraint abstraction into the type schema. We also need a form of judgements for groups of bindings. As you would expect the type of a group of bindings is just a set of type associations (i.e., a typing context) and the typing rules just collect the type associations and the corresponding constraint abstractions. In the rule Let

$$\frac{\Pi_0; \Gamma_0, \Gamma_1 \vdash \vec{b} : \Delta \textbf{ where } \vec{\phi} \quad \Pi_1; \Psi; \Gamma_2, \Gamma_3 \vdash e : \tau}{\Pi; \Psi; \Gamma_1, \Gamma_3 \vdash \texttt{let}\, \vec{b}\, \texttt{in}\, e : \tau} \quad \begin{array}{l} \mathrm{dom}(\Gamma_1, \Gamma_3) \cap \mathrm{dom}(\Delta) = \emptyset \\ \Pi \models \Delta \rightarrow^* \Gamma_0; \Gamma_2 \\ \Pi \models \Pi_0, \mathrm{let}\, \vec{\phi}\, \mathrm{in}\, \Pi_1 \end{array}$$

we first type the bindings which gives a context Δ which contains the type schemas associated with each binding. The first two side conditions ensures that the type schema $\chi_i{}^{\kappa_i'}_{\kappa_i}$ associated with each variable x_i in Δ is consistent with the type of each use of x_i. They also ensures that if x_i may be used more than once then κ_i and κ_i' must be constrained to ω. It is achieved as follows. If x_i occurs more than once in e and the right hand sides of $\vec{b}$ then x_i will also occur more than once in Γ_0, Γ_2. Thus the second side condition will ensure that κ_i and κ_i' is constrained to be ω. The typing of the bindings also gives a group of constraint abstraction $\vec{\phi}$. With the constraint abstraction we form the constraint let $\vec{\phi}$ in Π_1 which by the third side condition must be a consequence of the constraints in the conclusion of the rule.

3.8 Soundness

The soundness of our type system simply says that a well typed program is well annotated, i.e., when we run it in the abstract machine it does not go wrong.

Theorem 3.1
If $\Theta; \emptyset \vdash e : \tau$ and $\vartheta \models \Theta$ then $e\vartheta$ cannot go wrong.

The result is established by extending the type system to abstract machine configurations and then proving a subject reduction result which says that typings are preserved by transitions in the abstract machine. A very similar proof for the type system in [Gus98] is presented in full detail in [Gus99].

3.9 Inference Algorithm

As stated the type system is undecidable since it employs type polymorphic recursion. Our inference algorithm will therefore take a term which is explicitly typed in the underlying ordinary type system and can handle type polymorphic recursion if presented to it through the type annotations. It will first compute

a usage typing judgement which is principal with respect to the given typing judgement, i.e., every other usage typing judgement is an instance of the computed judgement if "stripping the annotations" from it yields the judgement in the underlying type system. The second phase of the algorithm then computes the best solution to the constraints in the principal judgement using the techniques described in a companion paper [GS01].

The time complexity of the algorithm is dominated by the cost of the constraint solving in the second phase. We can argue, as follows, that the time complexity of the second phase is $O(n^3)$ where n is the size of the explicitly typed term. Let the *skeleton* of the constraints be the constraints where all occurrences of inequality constraints of the form $\kappa_0 \leq \kappa_1$ have been removed. What remains are the binding occurrences of variables and all calls to constraint abstractions. By inspecting the typing rules we can see that the size of the skeleton of the constraints required to type a program is proportional to the size of the explicitly typed program. Moreover the number of free annotation variables in the constraints are proportional to the size of the program. From these facts and theorem 2 of [GS01] we can conclude that the complexity is $O(n^3)$ where n is the size of the typed program.

For a version of the analysis in this paper without usage-polymorphic recursion we have developed an algorithm based on *non-recursive* constraint abstractions with a worst case complexity of $O(n * m * t^2)$ where n is the size of the untyped lambda lifted version of the program, m is the size of the type of the largest set of (properly) mutually recursive definitions and t is the size of the largest instantiated type [Sve00]. Since m and t typically grow slowly or not at all with program size we expect that algorithm to scale up well in practice.

4 Related Work

There is a rich literature on analyses which aims at avoiding updates. See [Gus99] for a thorough overview. This work especially lends ideas from the type based approach by Turner, Wadler and Mossin [TWM95], and its followups by Gustavsson [Gus98] and Wansbrough and Peyton Jones [WPJ99]. Bounded polymorphism was proposed by Turner, Wadler and Mossin [TWM95] and the idea to use subtyping in usage analysis originates from the work by Faxén [Fax95] (the subtyping in his flow analysis and the directed edges in the post processing achieves the same effect as the subtyping in this paper) although it was independently proposed by Gustavsson [Gus98] and Wansbrough and Peyton Jones [WPJ99].

The analysis which seems to be closest in expressive power to ours is an analysis by Faxén based on an undecidable type based flow analysis [Fax97]. Due to the undecidable nature of the analysis his inference algorithm is not complete with respect to the type system. The algorithm is parametrised by a notion of finite name supply and the larger name-supply the better the algorithm

approximates the type system. The exact relationship between the different degrees of approximations computed by his algorithm and our type system is not clear to us.

The aim of this work is to make usage analysis scale up for large programs and in that respect it is most closely related to recent work by Wansbrough and Peyton Jones [WPJ00]. They have also observed that usage polymorphism is crucial for the accuracy of the analysis of large programs but they side-step the difficulties associated with bounded polymorphism. Instead they have a simple usage polymorphism where the quantified variables may not be constrained. This is achieved by an algorithm which eliminates inequality constraints prior to quantification by unifying constrained variables. The drawback of their approach is that as they refrain from using bounded polymorphism, they get an analysis which is rather inaccurate when it comes to data structures. Consider for example the following program fragment.

$$\ldots \text{map square (fromto 1 100)} \ldots$$

The spine of the list produced by *fromto* is consumed linearly by *map* but a type system with their simple usage polymorphism cannot discover it. The reason being that in a system with simple usage polymorphism the usage of the spine must be unified with the usage of the elements and in this case the elements are used more than once. In our system with bounded polymorphism the usage of the spine and the elements need only to constrain each other through an inequality constraint so we can deduce that the spine is used linearly although the elements are not. We believe that this situation is common enough in practice to have a significant effect on the accuracy of the analysis.

That the number of constraints explodes is a problem also for other type based program analyses with bounded polymorphism. In that respect our work is most closely related to the work by Faxén [Fax95], Mossin [Mos97] and Rehof and Fänhdrich [RF01]. Faxén and Mossin present inference algorithms for type based flow analyses which simplifies constraint sets to smaller but equivalent constraint sets. In their recent work on type based flow analysis Rehof and Fänhdrich uses *instantiation constraints* to represent constraints compactly and thus instantiation constraints plays a rôle similar to our constraint abstractions.

5 Conclusions and Future Work

We have presented a powerful and accurate type system for usage analysis with bounded usage polymorphism and subtyping. A key contribution is a new expressive form of constraints which allows constraints to be represented compactly through calls to *constraint abstractions*. In a companion paper [GS01] we show how to efficiently compute a least solution to constraints with constraint abstractions and we use this technique to obtain an $O(n^3)$ inference algorithm for our usage analysis, where n is the size of the explicitly typed program.

Acknowledgements We would like to thank David Sands and Makoto Takeyama for comments on this paper and Karl-Filip Faxén, Jakob Rehof and Keith Wansbrough for discussions on the relations to their work.

References

[BJ96] U. Boquist and T. Johnsson. The grin project: A highly optimising back end for lazy functional languages. In *Proc. of IFL'96, Bad Godesberg, Germany*. Springer Verlag LNCS 1268, 1996.

[BS96] E. Barendsen and S. Smetsers. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996.

[DHM95] D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In *proceedings of 2nd Static Analysis Symposium*, September 1995.

[Fax95] Karl-Filip Faxén. Optimizing lazy functional programs using flow inference. In *Proc. of SAS'95*, pages 136–153. Springer-Verlag, LNCS 983, September 1995.

[Fax97] Karl-Filip Faxén. *Analysing, Transforming and Compiling Lazy Functional Programs*. PhD thesis, Royal Institute of Technology, Sweden, June 1997.

[FW87] J. Fairbairn and S. Wray. TIM: A Simple, Lazy Abstract Machine to Execute Supercombinators. In *Proc. of FPCA'87*, pages 34–45. Springer Verlag LNCS 274, September 1987.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GS99] J. Gustavsson and D. Sands. A foundation for space-safe transformations of call-by-need programs. In *Proc. of HOOTS'99*, volume 26 of *ENTCS*. Elsevier, 1999.

[GS01] J. Gustavsson and J. Svenningsson. Constraint abstractions. In *Proc. of Second Symposium on Programs as Data Objects*, LNCS. Springer Verlag, 2001. To Appear.

[Gus98] J. Gustavsson. A Type Based Sharing Analysis for Update Avoidance and Optimisation. In *Proc. of ICFP'98*, pages 39–50, Baltimore, Maryland, September 1998.

[Gus99] J. Gustavsson. A Type Based Sharing Analysis for Update Avoidance and Optimisation. Licentiate thesis, May 1999.

[Lau93] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. of POPL'93*, Charleston, N. Carolina, 1993.

[LGH+92] J. Launchbury, A. Gill, J. Hughes, S. Marlow, S. L. Peyton Jones, and P. Wadler. Avoiding Unnecessary Updates. In J. Launchbury and P. M. Sansom, editors, *Functional Programming*, Workshops in Computing, Glasgow, 1992.

[Mar93] S. Marlow. Update Avoidance Analysis by Abstract Interpretation. In *Proc. 1993 Glasgow Workshop on Functional Programming*, Workshops in Computing. Springer–Verlag, 1993.

[Mog97] T. Mogensen. Types for 0, 1 or many uses. In *Proc. of IFL '97*, pages 112–122. Springer-Verlag, LNCS 1467, September 1997.

[Mos97] C. Mossin. *Flow Analysis of Typed Higher-Order Programs (Revised Version)*. PhD thesis, University of Copenhagen, Denmark, August 1997.

[PJ92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, July 1992.

[PJPS96] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proc. of ICFP'96*, pages 1–12. ACM, May 1996.

[PS00] R. Peña and C. Segura. Non-determinism analysis in a parallel-functional language. In *Proceedings of the 12th International Workshop of Functional Languages*, LNCS, pages 1–18. Springer-Verlag, LNCS 2011, September 2000.

[RF01] Jakob Rehof and Manuel Fändrich. Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In *Proceedings of 2001 Symposium on Principles of Programming Languages*, 2001. To appear.

[Ses91] P. Sestoft. *Analysis and Efficient Implementation of Functional Programs*. PhD thesis, DIKU, University of Copenhagen, Denmark, October 1991.

[Ses97] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[Sve00] Josef Svenningsson. An efficient algorithm for a sharing analysis with polymorphism and subtyping. Masters thesis, June 2000.

[TJ94] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2), 1994.

[TWM95] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proc. of FPCA*, La Jolla, 1995.

[WPJ98] Keith Wansbrough and Simon Peyton Jones. Once Upon a Polymorphic Type. Technical Report TR-1998-19, Department of Computing Science, University of Glasgow, December 1998.

[WPJ99] Keith Wansbrough and Simon Peyton Jones. Once Upon a Polymorphic Type. In *Proc. of POPL'99*, January 1999.

[WPJ00] Keith Wansbrough and Simon Peyton Jones. Simple Usage Polymorphism. In *ACM SIGPLAN Workshop on Types in Compilation*, September 2000.

Paper V

Constraint Abstractions

Constraint Abstractions

Jörgen Gustavsson Josef Svenningsson

Abstract

Many type based program analyses with subtyping, such as flow analysis, are based on inequality constraints over a lattice. When inequality constraints are combined with polymorphism it is often hard to scale the analysis up to large programs. A major source of inefficiency in conventional implementations stems from computing substitution instances of constraints. In this paper we extend the constraint language with *constraint abstractions* so that instantiation can be expressed directly in the constraint language and we give a cubic-time algorithm for constraint solving. As an application, we illustrate how a flow analysis with flow subtyping, flow polymorphism and flow-polymorphic recursion can be implemented in $O(n^3)$ time where n is the size of the explicitly typed program.

1 Introduction

Constraints are at the heart of many modern program analyses. These analyses are often implemented by two stages. The first stage collects constraints in an appropriate constraint language and the second stage finds a solution (usually the least) to the constraints. If the constraints are collected through a simple linear time traversal over the program yielding a linear amount of constraints the first phase can hardly constitute a bottleneck. But often the constraints for a program point are computed by performing a non constant-time operation on constraints collected for another part of the program. Notable examples, and the motivation for this work, are analyses which combine subtyping and polymorphism. There, typically, the constraints for a call to a polymorphic function f are a *substitution instance* of the constraints for the body of f. For these analyses, to naïvely collect constraints typically leads to unacceptable performance. Consider, for example, how we naïvely could collect the constraints for a program of the following form.

$$
\begin{array}{l}
\texttt{let}\ f_0 = \ldots \\
\texttt{in}\ \texttt{let}\ f_1 = \ldots f_0 \ldots f_0 \\
\quad \texttt{in}\ \texttt{let}\ \ldots \\
\qquad \texttt{in}\ \texttt{let}\ f_n = \ldots f_{n-1} \ldots f_{n-1} \ldots \\
\qquad\quad \texttt{in}\ \ldots f_n \ldots f_n \ldots
\end{array}
$$

We first collect the constraints for the polymorphic function f_0. Then for the two calls to f_0 in the body of f_1, we compute two different substitution instances of the constraints from the body of f_0. As a result the number of constraints for f_1 will be at least twice as many as those for f_0. Thus, the number of resulting constraints grows exponentially in the call depth n (even if the underlying types are small). In analyses which combine subtyping and polymorphic recursion, and rely on a fixed point iteration, this effect may show up in every step of the iteration and thus the constraints may grow exponentially in the number of required iterations. We can drastically reduce the number of constraints if we can simplify the constraints to fewer but equivalent constraints. It is therefore no surprise that lots of work has been put into techniques for how to simplify constraints [FM89, Cur90, Kae92, Smi94, EST95, Pot96, TS96, FA96, AWP97, Reh97, FF97].

Another approach is to make the constraint language more powerful so that constraints can be generated by a simple linear time traversal over the program. This can be achieved by *making substitution instantiation a syntactic construct in the constraint language*. But when we make the constraint language more powerful we also make constraint solving more difficult. So is this a tractable approach? The constraint solver could of course just perform the delayed operations and then proceed as before. But can one do better? The answer, of course, depends on the constraint language in question.

In this paper we consider a constraint language with simple inequality constraints over a lattice. Such constraints show up in several type based program analyses such as flow analyses, e.g., [Mos97], binding time analyses, e.g., [DHM95], usage analyses, e.g., [TWM95], points-to-analyses, e.g., [FFA00] and uniqueness type systems [BS96]. We extend this simple constraint language with *constraint abstractions* which allow the constraints to compactly express substitution instantiation.

The main result of this paper is a constraint solving algorithm which computes least solutions to the extended form of constraints in cubic time. We have used this expressive constraint language to formulate usage-polymorphic usage analyses with usage subtyping [Sve00, GS00] and an algorithm closely related to the one in this paper is presented in the second author's Master's thesis [Sve00] ([GS00] focuses on the usage type system and no constraint solving is presented). In this paper, as another example, we show how the constraint language can be used to yield a cubic algorithm for Mossin's polymorphic flow analysis with flow subtyping and flow-polymorphic recursion [Mos97]. This is a significant result – the previously published algorithm, by Mossin, is $O(n^8)$. Independently, Fähndrich and Rehof [RF01] have given an algorithm for Mossin's flow analysis based on *instantiation constraints* which is also $O(n^3)$. We will take a closer look an the relationship of their algorithm and ours in section 4.

1.1 Outline

The rest of this article is organised as follows. In section 2 we introduce our constraint language and give the semantics. In section 3 we present our constraint solving algorithm, its implementation and computational complexity. Section 4 discusses related work and section 5 concludes. In appendix A we illustrate how the constraint language can be used in a flow analysis. In appendix B we give the proof of Theorem 3.6.

2 Constraints

In this section we will first introduce the underlying constraints language that we consider in this paper, and then extend the constraint language with constraint abstractions which can express substitution instantiation. The *atomic* constraints we consider are inequality constraints of the form

$$a \leq b$$

where a and b are taken from an countably infinite set of variables. The constraint language also contains the trivially true constraint, conjunction of constraints and existential quantification as given by the following grammar.

$$\begin{array}{lrcl} \text{Atomic Constraints} & A & ::= & a \leq b \\ \text{Constraint Terms} & M, N & ::= & A \mid \top \mid M \wedge N \mid \exists a.M \end{array}$$

These kinds of constraints show up in several different type based program analyses such as, for example, flow analysis, e.g., [Mos97] which we will use as our running example. The constraints arise from the use of subtyping between flow types - i.e., types annotated with flow information.

Depending on the application, the constraints can be interpreted in different domains. For example, for flow analysis we can interpret the constraints in a lattice of finite sets of labels with subset as the ordering.

Definition 2.1
We interpret a constraint term in a lattice $\mathcal{L}$, with a bottom element and the ordering $\sqsubseteq$, by defining the notion of a model of a constraint term. Let θ range over mappings from variables into $\mathcal{L}$. Then $\theta \models M$, read as θ is a model of M, is defined inductively by the following rules.

$$\frac{\theta(a) \sqsubseteq \theta(b)}{\theta \models a \leq b} \qquad \frac{}{\theta \models \top} \qquad \frac{\theta \models M \qquad \theta \models N}{\theta \models M \wedge N} \qquad \frac{\theta[a := d] \models M}{\theta \models \exists a.M}\, d \in \mathcal{L}$$

Given a constraint term one is usually interested in finding its optimal model (usually the least) given a fixed assignment of some of the variables. For example, in flow analysis some of the variables in the constraint term correspond to points in the program where values are produced, often referred to as the *sources*

of flow. Other variables correspond to points in the program where values are consumed, often referred to as the *targets* of flow. The existentially quantified variables correspond to the flow annotations on intermediate flow types. To find the flow from the sources to the targets we can fix an assignment for the source variables (usually by associating a unique label l to each source and interpret it as the singleton set $\{l\}$) and compute the least model which respects this assignment. For this simple constraint language it is easy to compute least solutions (it can be seen as a transitive closure problem) in $O(n^3)$ time, where n is the number of variables. [1]

2.1 Constraint abstractions

When subtyping is combined with polymorphism the need to compute substitution instances of constraint terms arise. We will build this operation into our constraint language through the means of constraint abstractions.

Constraint Abstraction Variables	f, g, h		
Constraint Abstractions	F	$::=$	$f\ \vec{a} = M$

A constraint abstraction $f\ \vec{a} = M$ can be seen simply as a function which when applied to some variables $\vec{b}$ returns $M[\vec{a} := \vec{b}]$. Constraint abstractions are introduced by a let-construct reminiscent of let-constructs in functional languages, and are also called in the same way. The complete grammar of the extended constraint language is as follows.

Atomic Constraints	A	$::=$	$a \leq b$
Constraint Terms	M, N	$::=$	$A \mid \top \mid M \wedge N \mid \exists a.M \mid$
			$\mathtt{let}\ \{\vec{F}\}\ \mathtt{in}\ M \mid f\ \vec{a}$
Constraint Abstractions	F	$::=$	$f\ \vec{a} = M$

We will write $\mathrm{FV}(M)$ for the free variables of M and $\mathrm{FAV}(M)$ for the free abstraction variables of M. We will identify constraint terms up to α-equivalence, that is the renaming of bound variables and bound abstraction variables. In $\mathtt{let}\ \{\vec{F}\}\ \mathtt{in}\ M$ the constraint abstraction variables defined by $\vec{F}$ are bound both in M and in the bodies of $\vec{F}$ so our lets are mutually recursive. Consequently the variables defined by $\vec{F}$ must be distinct. We will use Γ to range over sets of constraint abstractions where the defined variables are distinct, and we will denote the addition of a group of distinct constraint abstractions $\vec{F}$ to Γ by juxtaposition: $\Gamma\{\vec{F}\}$. We will say that a group of constraint abstractions $\vec{F}$ is *garbage* in $\mathtt{let}\ \Gamma\{\vec{F}\}\ \mathtt{in}\ M$ if we can remove the abstractions without causing bound abstraction variables to become free. Recursive constraint abstractions goes beyond just expressing a delayed substitution instantiation. It also allows us to express a fixed-point calculation in a very convenient way. We will

[1] For a lattice where binary least upper bounds can be computed in constant time (for example a two point lattice) the least solution can be computed in $O(n^2)$ time.

make use of this in the flow analysis in appendix A to express flow-polymorphic recursion.

To give a semantics to the extended constraint language we need to define the notion of a model of a constraint term in the context of a set of constraint abstractions Γ.

Definition 2.2
In a lattice $\mathcal{L}$, with a bottom element and with the ordering $\sqsubseteq$, we define $\theta; \Gamma \models M$ coinductively by the following rules (we follow the notational convention of Cousot and Cousot [CC92] to mark the rules with a "$-$" to indicate that it is a coinductive definition).

$$-\frac{}{\theta; \Gamma \models a \leq b} \quad \theta(a) \sqsubseteq \theta(b) \qquad -\frac{\theta; \Gamma \models M \qquad \theta; \Gamma \models N}{\theta; \Gamma \models M \wedge N}$$

$$-\frac{}{\theta; \Gamma \models \top} \qquad -\frac{\theta[a := d]; \Gamma \models M \qquad d \in \mathcal{L}}{\theta; \Gamma \models \exists a.M \qquad a \notin \mathrm{FV}(\Gamma)}$$

$$-\frac{\theta; \Gamma\{\vec{F}\} \models M}{\theta; \Gamma \models \mathtt{let}\ \{\vec{F}\}\ \mathtt{in}\ M} \qquad -\frac{\theta; \Gamma\{f\ \vec{a} = M\} \models M[\vec{a} := \vec{b}]}{\theta; \Gamma\{f\ \vec{a} = M\} \models f\ \vec{b}}$$

The definition needs to be coinductive to cope with recursive constraint abstractions. The coinductive definition expresses the intuitive concept that such constraint abstractions should be "unfolded infinitely". When it is not clear from the context we will write $\theta; \Gamma \models_{\mathcal{L}} M$ to make explicit which lattice we consider. We will say that N is a *consequence* of M, written $M \models N$, iff for every $\mathcal{L}$, θ, Γ, if $\theta; \Gamma \models_{\mathcal{L}} M$ then $\theta; \Gamma \models_{\mathcal{L}} N$. We will write $M \Leftrightarrow N$ iff $M \models N$ and $N \models M$.

In definitions throughout this paper we will find it convenient to work with *constraint term contexts*. A constraint term context is simply a constraint term with a "hole" analogous to term contexts used extensively in operational semantics.

$$\text{Constraint Term Contexts} \quad C ::= [\cdot] \mid C \wedge M \mid M \wedge C \mid \exists a.C \mid \mathtt{let}\ \Gamma\ \mathtt{in}\ C \mid \mathtt{let}\ \Gamma\{f\ \vec{a} = C\}\ \mathtt{in}\ M$$

We will write $C[M]$ to denote the filling of the hole in C with M. Hole filling may capture variables. We will write $\mathrm{CV}(C)$ for the variables that may be captured when filling the hole. We will say that the hole in C is *live* if the hole does not occur in a constraint abstraction which is garbage. Our first use of constraint term contexts is in the definition of the *free live atomic constraints* of a constraint term.

Definition 2.3
The set of free live atomic constraints of a constraint term M, denoted $\mathrm{LIVE}(M)$, is defined as follows.

$$\mathrm{LIVE}(M) = \{A \mid M \equiv C[A], \mathrm{FV}(A) \cap \mathrm{CV}(C) = \emptyset \text{ and the hole in } C \text{ is live.}\}$$

We will use $\mathrm{LIVE}(M)$ in definitions where we need to refer to the atomic subterms of M but want to exclude those which occur in constraint abstractions which are garbage and thus never will be "called" by the models relation. Note that all syntactically live constraint abstractions are semantically live since they are all "called" by the models relation.

Another use of constraint term contexts is in the statement of the following unwinding lemma.

Lemma 2.4
If $\mathrm{FV}(M) \cap \mathrm{CV}(C) = \emptyset$ then

$$\mathtt{let}\ \Gamma\{f\ \vec{a} = M\}\ \mathtt{in}\ C[f\ \vec{b}] \Leftrightarrow \mathtt{let}\ \Gamma\{f\ \vec{a} = M\}\ \mathtt{in}\ C[M[\vec{a} := \vec{b}]]$$

This lemma is necessary, and is the only difficulty, when proving the subject reduction property of the usage analysis in [GS00] and the flow analysis in appendix A. The premise $\mathrm{FV}(M) \cap \mathrm{CV}(C) = \emptyset$ is there to ensure that no inadvertent name capture takes place and it can always be fulfilled by an α-conversion. In the remainder of this paper we will leave this condition on unwindings implicit.

3 Solving Constraints

As we discussed in the previous section we are interested in finding the least model of a constraint term given a fixed assignment of some of the variables. In this section we will present an algorithm for this purpose for our constraint language. The algorithm is based on a rewrite system which rewrites constraint terms to equivalent but more informative ones. Every rewrite step adds an atomic constraint to the constraint term and the idea is that when the rules have been applied exhaustively then enough information is explicit in the term so that the models can be constructed easily.

Definition 3.1
We define the rewrite relation $\rightarrow$ as the compatible closure of the relation $\mapsto$ defined by the clauses in figure 1.

Here we provide some explanation of the rewrite rules. The first rule,

1. if $a \leq b, b \leq c \in \mathrm{LIVE}(M)$ then

   $\exists b.M \ \mapsto\ \exists b.M \wedge a \leq c$

is a simple transitivity rule. If $a \leq b$ and $b \leq c$ are free live atomic subterms of M we may simply add the constraint $a \leq c$. Note that the rule requires a and c to be in scope at the binding occurrence of b. As a result we cannot, for example, perform the rewrite

$$\exists a.\exists b.(a \leq b) \wedge (\exists c.b \leq c) \ \rightarrow\ \exists a.\exists b.(a \leq b) \wedge (\exists c.b \leq c \wedge a \leq c)$$

1. if $a \leq b, b \leq c \in \mathrm{LIVE}(M)$ then

   $\exists b.M \;\mapsto\; \exists b.M \wedge a \leq c$

2. if $A \in \mathrm{LIVE}(M)$, and, for some i, $a_i \in \mathrm{FV}(A)$ then

   $$\begin{array}{ll} \mathtt{let}\,\Gamma\{f\,\vec{a} = M\} & \mapsto\; \mathtt{let}\,\Gamma\{f\,\vec{a} = M\} \\ \mathtt{in}\,C[f\,\vec{b}] & \quad\;\; \mathtt{in}\,C[f\,\vec{b} \wedge A[\vec{a} := \vec{b}]] \end{array}$$

3. if $A \in \mathrm{LIVE}(C[f\,\vec{b}])$, and, for some i, $a_i \in \mathrm{FV}(A)$ then

   $$\begin{array}{ll} \mathtt{let}\,\Gamma\{f\,\vec{a} = C[f\,\vec{b}]\} & \mapsto\; \mathtt{let}\,\Gamma\{f\,\vec{a} = C[f\,\vec{b} \wedge A[\vec{a} := \vec{b}]]\} \\ \mathtt{in}\,M & \quad\;\; \mathtt{in}\,M \end{array}$$

4. if $A \in \mathrm{LIVE}(M)$, and for some i, $a_i \in \mathrm{FV}(A)$ then

   $$\begin{array}{l} \mathtt{let}\,\Gamma\{f\,\vec{a} = M\}\{g\,\vec{c} = C[f\,\vec{b}]\} \\ \mathtt{in}\,M \\ \mapsto \\ \mathtt{let}\,\Gamma\{f\,\vec{a} = M\}\{g\,\vec{c} = C[f\,\vec{b} \wedge A[\vec{a} := \vec{b}]]\} \\ \mathtt{in}\,M \end{array}$$

Figure 1: Rewrite rules

which adds $a \leq c$ although it would make perfect sense. The reason is simply that at the binding occurrence of b, c is not in scope. The purpose of the restriction on the transitivity rule is an important one. It reduces the number of rewrite steps due to transitivity by taking advantage of scoping information. The second rule

2. if $A \in \mathrm{LIVE}(M)$, and, for some i, $a_i \in \mathrm{FV}(A)$ then

   $$\begin{array}{ll} \mathtt{let}\,\Gamma\{f\,\vec{a} = M\} & \mapsto\; \mathtt{let}\,\Gamma\{f\,\vec{a} = M\} \\ \mathtt{in}\,C[f\,\vec{b}] & \quad\;\; \mathtt{in}\,C[f\,\vec{b} \wedge A[\vec{a} := \vec{b}]] \end{array}$$

allows us to unwind an atomic constraint. Note that at least one of the variables in A must be bound by the abstraction. The restriction is there to prevent rewrite steps which would not be useful anyway. The two last rules are similar to the second rule but deal with unwinding in mutually recursive constraint abstractions. A key property of the rewrite rules is that they lead to equivalent constraint terms.

Lemma 3.2
If $M \mapsto N$ then $M \Leftrightarrow N$

The property is easy to argue for the transitivity rule. For the second rule it follows from the unwinding property (Lemma 2.4). The two last rules rely on similar unwinding properties for unwinding in mutually recursive constraint abstractions.

3.1 Normal forms

Intuitively a constraint term is in normal form when the rules in figure 1 have been applied exhaustively. But nothing stops us from performing rewrite steps which just add new copies of atomic constraints which are already in the constraint term. We can of course do this an arbitrary number of times creating a sequence of terms which are different but "essentially the same". To capture this notion of essentially the same we define a congruence which equates terms which are equal up to copies of atomic constraints.

Definition 3.3
We define $\sim$ as the reflexive, transitive, symmetric and compatible closure of the following clauses.

$$\text{(i) } A \wedge A \sim A \qquad \text{(ii) } M \wedge \top \sim M \qquad \text{(iii) } \top \wedge M \sim M$$
$$\text{(iv) if } \mathrm{FV}(A) \cap \mathrm{CV}(C) = \emptyset \text{ and the hole in } C \text{ is live then } C[A] \sim C[\top] \wedge A$$

Rewriting commutes with $\sim$ so we can naturally extend $\rightarrow$ to equivalence classes of $\sim$. With the help of $\sim$ we can define the notion of a *productive* rewrite step $M \rightsquigarrow N$ which is a rewrite step which adds a new atomic constraint.

Definition 3.4
$M \rightsquigarrow N$ iff $M \rightarrow N$ and $M \not\sim N$.

Finally we arrive at our definition of *normal form* up to productive rewrite steps.

Definition 3.5
M is in normal form iff $M \not\rightsquigarrow$.

The main technical theorem in this paper is that when a constraint term with no free constraint abstraction variables is in normal form then the models of the constraint term are exactly characterised by the free live atomic constraints of the constraint term.

Theorem 3.6
If M is in normal form and $\mathrm{FAV}(M) = \emptyset$ then $\theta; \emptyset \models M$ iff $\theta \models \mathrm{LIVE}(M)$

Given a constraint term M and a fixed assignment of some of the variables we can find its least model as follows. First we find an equivalent constraint term N

in normal form. Then we extract the free live atomic constraints of the normal form which exactly characterises the models of N and M. Since LIVE(N) is just a set of atomic constraints we can then proceed with any standard method, such as computing the transitive closure. The proof of Theorem 3.6 can be found in appendix B. The key component of the proof is the application of two key properties of unwindings of normal forms. The first property is that normal forms are preserved by unwindings.

Lemma 3.7
If $\texttt{let}\,\Gamma\{f\,\vec{a} = M\}\,\texttt{in}\,C[f\,\vec{b}]$ is in normal form then the unwinding $\texttt{let}\,\Gamma\{f\,\vec{a} = M\}\,\texttt{in}\,C[M[\vec{a} := \vec{b}]]$ is in normal form.

The lemma guarantees normal forms of arbitrary unwindings of a normal form which we need because of the coinductive definition of $\theta; \Gamma \models M$. The second property is that unwinding of a normal form does not change the free live atomic constraints of the constraint term.

Lemma 3.8
If $\texttt{let}\,\Gamma\{f\,\vec{a} = M\}\,\texttt{in}\,C[f\,\vec{b}]$ is in normal form then

$$\text{LIVE}(\texttt{let}\,\Gamma\{f\,\vec{a} = M\}\,\texttt{in}\,C[f\,\vec{b}]) = \text{LIVE}(\texttt{let}\,\Gamma\{f\,\vec{a} = M\}\,\texttt{in}\,C[M[\vec{a} := \vec{b}]])$$

3.2 Computing Normal Forms

Given a constraint term M, we need to compute an equivalent term in normal form. Our algorithm relies on a representation of equivalence classes of terms with respect to $\sim$ and computes sequences of the form

$$M_0 \rightsquigarrow M_1 \rightsquigarrow M_2 \rightsquigarrow \ldots.$$

The termination of the algorithm is ensured by the following result.

Lemma 3.9
There is no infinite sequence of the form given above.

Proof 3.10 (Sketch)
Let n be the number of variables (free and bound) in M_0. Note that the number of variables remain constant in each step. Thus the number of unique atomic constraints that can be added to M is bounded by n^2. Since every productive rewrite step introduces a new atomic constraint the number of steps is bounded by n^2.

When given a constraint term as input, our algorithm first marks all atomic constraints. These marked constraints can be thought of as a work list of atomic constraints to consider. The algorithm then unmarks the constraints one by one and performs all productive rewrite steps which only involve atomic constraints

which are not marked. The new atomic constraints which are produced by a rewrite step are initially marked. The algorithm maintains the following invariant: the term obtained by replacing the marked terms with $\top$ is in normal form. The algorithm terminates with a normal form when no atomic constraints remain marked. The pseudo code for this algorithm is given below.

Algorithm 3.11

1. Mark all atomic constraints.
2. If there are no remaining marked constraints then stop otherwise pick a marked atomic constraint and unmark it.
3. Find all productive redexes which involve the unmarked constraint and perform the corresponding rewrite steps. Let the added atomic constraints be marked.
4. Go to step 2.

3.3 Data Structures

The efficiency of the algorithm relies on maintaining certain data structures. In step 3 of the algorithm we use data structures such that we can solve the following two problems:

1. find all redexes we need to consider in time proportional to the number of such, and
2. decide in constant time whether a redex is productive.

We can solve the first problem if we maintain, for every existentially bound variable b,

- a list of all a in scope at the point where b is bound, such that $a \leq b$ is an unmarked atomic constraint in the term.
- a list of all c in scope at the point where b is bound, such that $b \leq c$ is an unmarked atomic constraint in the term.

With this information we can easily list all transitivity-redexes we need to consider in step 3, in time proportional to the number of redexes. When we unmark a constraint we can update the data structure in constant time.

For the second problem, to decide in constant time whether a redex is productive, we need to decide, in constant time, whether the atomic constraint to be added already exists in the term. We can achieve this by a n times n bit-matrix where n is the number of variables (free and bound) in the constraint term. If $a \leq b$ is in the term then the entry in the matrix for (a, b) is 1 and 0 otherwise. This is sufficient for the complexity argument in the next section but in practice we use a refined data structure which we describe in section 3.5.

3.4 Complexity

The cost of the algorithm is dominated by the operations performed by step 3, which searches for productive redexes. The cost is proportional to the number of redexes (productive or non-productive) considered and each redex in the final normal form is considered exactly once in step 3. Thus the cost of step 3 is proportional to the number of redexes in the final normal form. An analysis of the maximum number of redexes gives the following.

- The maximum number of transitivity-redexes is, for each existentially quantified variable a, the square of the number of variables in scope at the point where a is bound.
- The maximum number of unwind-redexes is, for each variable a bound in a constraint abstraction f, two times the number of variables in scope at the point where a is bound times the number of calls to f.

A consequence of this analysis is the complexity result we are about to state. Let the *skeleton* of a constraint term be the term where all occurrences of atomic constraints, and the trivially true constraint have been removed. What remains are the binding occurrences of variables and all calls to constraint abstractions. Now, for a constraint term M, let n be the size of the skeleton of M plus the number of free variables of M. The complexity of the algorithm can be expressed in terms of n as follows.

Theorem 3.12
The normal form can be computed $O(n^3)$ time.

3.5 Refined Data Structure

The cost of initialising the bit-matrix described in section 3.3 is dominated by the cost of step 3 in the algorithm but we believe that in practice the cost of initialising the matrix may be significant. Also the amount of memory required for the matrix is quite substantial and many entries in the matrix would be redundant since the corresponding variables have no overlapping scope. Below we sketch a refined approach based on this observation which we believe will be important in practice. We associate a natural number, index(a), with every variable a. We assign the natural number as follows. First we choose an arbitrary order for all the free variables and bind them existentially, in this order, at top level. Then we assign to each variable the lexical binding level of the variable. For example, in $\exists a.(\exists b.M) \wedge (\exists c.N)$ we assign 0 to a, 1 to b and c, and so on. Note that the number we assign to each variable is unique within the scope of the variable. Given this we have the following data structures. For every variable b,

- a set of all a such that index(a) $\leq$ index(b) and $a \leq b$ is an atomic constraint (marked or unmarked) in the term.

- a set of all c such that index(c) $\leq$ index(b) and $b \leq c$ is an atomic constraint (marked or unmarked) in the term.

The sets have, due to scoping, the property that, for any two distinct elements a and b, index(a) is distinct from index(b). Thus the sets can be represented by bit-arrays, indexed by index(a) so that set membership can be decided in constant time. Now, to decide whether an atomic constraint $a \leq b$ is in the constraint becomes just set membership in the appropriate set.

4 Related Work

The motivation for this paper is to reduce the cost of the combination of subtyping and polymorphism and in this respect it is related to numerous papers on constraint simplification techniques
[FM89, Cur90, Kae92, Smi94, EST95, Pot96, TS96, FA96, AWP97, Reh97, FF97]. Our work is particularly related to the work by Dussart, Henglein and Mossin on binding-time analysis with binding-time-polymorphic recursion [DHM95] where they use constraint simplification techniques in combination with a clever fixed-point iteration to obtain a polynomial time algorithm. In his thesis Mossin applied these ideas to show that a flow analysis with flow-polymorphic recursion can be implemented in polynomial time [Mos97]. Our flow analysis in appendix A, that we give as an example of how constraint abstractions can be used, is based on this flow analysis. A consequence of the complexity of our constraint solving algorithm is that the analysis can be implemented in $O(n^3)$ time where n is the size of the explicitly type program. This is a substantial improvement over the algorithm by Mossin which is $O(n^8)$ [2] [Mos97].

To represent instantiation in the constraint language is not a new idea. It goes back at least to Henglein's work on type-polymorphic recursion [Hen93] where he uses *semiunification constraints* to represent instantiation. Although constraint abstractions and semiunification constraints may have similar applications they are inherently different: Semiunification constraints are inequality constraints of the form $A \leq B$ which constrains the (type) term B to be an instance of A by an *unknown* substitution. In contrast, a call to a constraint abstraction denotes a *given instance* of *the constraints* in the body of the abstraction.

Closely related to our work is the recent work by Rehof and Fähndrich [RF01] where they also give an $O(n^3)$ algorithm for Mossin's flow analysis. The key idea in their and our work is the same – to represent substitution instantiation in the constraints by extending the constraint language. However, the means are

[2] In his thesis Mossin states that he believes that the given algorithm can be improved. In fact an early version of [DHM95] contained a $O(n^3)$ algorithm for binding-time analysis but it was removed from the final version since its correctness turned out to be non-trivial (personal communication with Fritz Henglein).

not the same. Where we use constraint abstractions they use *instantiation constraints*, a form of inequality constraints similar to semiunification constraints but labelled with an instantiation site and a polarity. They compute the flow information from the constraints through an algorithm for *Context-Free Language (CFL) reachability* [Rep97, MR00]. A key difference between constraint abstractions and instantiation constraints is that constraint abstractions offer more structure and a notion of local scope whilst in the work by Rehof and Fähndrich all variables scope over the entire set of constraints. Our algorithm takes advantage of the scoping in an essential way. Firstly, we do not add any edges between variables that have no common scope and secondly the scoping comes into the restriction of our transitivity rule and the unwind rules. Although the scoping does not improve the asymptotic complexity in terms of the size of the explicitly typed program it shows up in the more fine-grained complexity argument leading to the cubic bound (see section 3.4) and it is essential for the refined data structures we sketch in section 3.5. Constraint abstractions also offer a more subjective advantage – the additional structure of constraint abstractions enforces many useful properties. As a result we think it will be easy to use constraint abstractions in a wide range of type based analyses and we think that constraint abstractions will not lead to any additional difficulties when establishing the soundness of the analyses.

We have previously used constraint abstraction to formulate usage-polymorphic usage analyses with usage subtyping [Sve00, GS00] and an algorithm closely related to the one in this paper is presented in the second authors masters thesis [Sve00] ([GS00] focuses on the usage type system and no constraint solving is presented).

5 Conclusions and Future Work

In this paper we have shown how a constraint language with simple inequality constraints over a lattice can be extended with constraint abstractions which allow the constraints to compactly express substitution instantiation. The main result of this paper is a constraint solving algorithm which computes least solutions to the extended form of constraints in cubic time. In [GS00] we have used this expressive constraint language to formulate a usage-polymorphic usage analyses with usage subtyping and usage-polymorphic recursion and in an appendix to this paper we demonstrate how the extended constraint language can be used to yield a cubic algorithm for Mossin's polymorphic flow analysis with flow subtyping and flow polymorphic recursion [Mos97]. We believe that our approach can be applied to a number of other type based program analyses such as binding time analyses, e.g., [DHM95], points-to-analyses, e.g., [FFA00] and uniqueness type systems [BS96].

An interesting possibility for future work is to explore alternative constraint solving algorithms. The current algorithm has a rather compositional character

in that, it rewrites the body of a constraint abstraction without considering how it is called. In [Sve00] we describe an algorithm where the different calls to a constraint abstraction lead to rewrites inside the abstraction. The algorithm can in this way take advantage of global information (it can be thought of as a form of caching) which yields a interesting finer grained complexity characterisation. The algorithm in [Sve00] is however restricted to *non-recursive* constraint abstractions and it is not clear whether the algorithm can be extended to recursive constraint abstractions (although we believe so). Another opportunity for future work is to investigate whether constraint abstractions can be a useful extension for other underlying constraint languages. Constraint abstraction could also possibly be made more powerful by allowing constraint abstractions to be passed as parameters to constraint abstractions (i.e., making them higher order). Finally a practical comparison with Mossin's algorithm and the algorithm by Rehof and Fähndrich remains to be done. The outcome of such a comparison is not clear to us.

Acknowledgements. We would like to thank our supervisor David Sands for his support and the anonymous referees for their useful comments.

References

[AWP97] A. Aiken, E. Wimmers, and J. Palsberg. Optimal representation of polymorphic types with subtyping. In *Proceedings TACS'97 Theoretical Aspects of Computer Software*, pages 47–77. Springer Lecture Notes in Computer Science, vol. 1281, September 1997.

[BS96] E. Barendsen and S. Smetsers. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996.

[CC92] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Conference Record of the Ninthteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–94, Albuquerque, New Mexico, January 1992. ACM Press, New York, NY.

[Cur90] P. Curtis. Constrained qualification in polymorphic type analysis. Technical Report CSL-09-1, Xerox Parc, February 1990.

[DHM95] D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In *proceedings of 2nd Static Analysis Symposium*, September 1995.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings OOPSLA'95*, 1995.

[FA96] M. Fändrich and A. Aiken. Making set-constraint program analyses scale. In *Workshop on Set Constraints*, 1996.

[Fax95] Karl-Filip Faxén. Optimizing lazy functional programs using flow inference. In *Proc. of SAS'95*, pages 136–153. Springer-Verlag, LNCS 983, September 1995.

[FF97] C. Flanagan and M. Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*. ACM, June 1997.

[FFA00] J. Foster, M. Fändrich, and A. Aiken. Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C. In *Proceedings of 2000 Static Analysis Symposium*, June 2000.

[FM89] Y. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *Proceedings of Int'l J't Conf. on Theory and Practice of Software Development*, pages 167–183. Springer-Verlag, March 1989.

[GS00] J. Gustavsson and J. Svenningsson. A usage analysis with bounded usage polymorphism and subtyping. In *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, pages 140–157. Springer-Verlag, LNCS 2011, September 2000.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[Kae92] Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *1992 ACM conference on LISP and Functional Programming*, pages 193–204, San Francisco, CA, 1992. ACM Press.

[Mos97] C. Mossin. *Flow Analysis of Typed Higher-Order Programs (Revised Version)*. PhD thesis, University of Copenhagen, Denmark, August 1997.

[MR00] David Melski and Thomas Reps. Interconvertibility of a class of set constraints and context-free language reachability. *Theoretical Computer Science*, 248, November 2000.

[Pot96] F. Pottier. Simplifying subtyping constraints. In *Proceedings ICFP'97, International Conference on Functional Programming*, pages 122–133. ACM Press, May 1996.

[Reh97] J. Rehof. Minimal typings in atomic subtyping. In *Proceedings POPL'97, 24th ACM SIGPLAN-SIGACT Symposium on Principles*

of Programming Languages, pages 278–291, Paris, France, January 1997. ACM.

[Rep97] T. Reps. Program analysis via graph reachability. In *Proc. of ILPS '97*, pages 5–19. Springer-Verlag, October 1997.

[RF01] Jakob Rehof and Manuel Fändrich. Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In *Proceedings of 2001 Symposium on Principles of Programming Languages*, 2001.

[Smi94] G. S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.

[Sve00] Josef Svenningsson. An efficient algorithm for a sharing analysis with polymorphism and subtyping. Masters thesis, June 2000.

[TS96] V. Trifonov and S. Smith. Subtyping constrained types. In *Proceedings SAS'96, Static Analysis Symposium*, pages 349–365, Aachen, Germany, 1996. Springer Verlag.

[TWM95] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proc. of FPCA*, La Jolla, 1995.

A Flow Analysis

In this appendix we illustrate how constraint abstractions can be used in practice. As an example, we briefly present a flow-polymorphic type based flow analysis with flow-polymorphic recursion. For another example see [GS00] where constraint abstractions are used in usage analysis. The flow analysis is based on the flow analysis by Mossin [Mos97] but we use our extended constraint language with constraint abstractions. A similar analysis, but without polymorphic recursion, is given by Faxén [Fax95]. For simplicity we restrict ourself to a simply typed functional language. To extend the analysis to a language with a Hindley-Milner style type system is not difficult. See for example [Fax95]. A key result is that the analysis can be implemented in $O(n^3)$ time where n is the size of the explicitly typed program which is a substantial improvement over the algorithm by Mossin which is $O(n^8)$ [Mos97].

The aim of flow analysis is to statically compute an approximation to the flow of values during the execution of a program. To be able to pose flow questions we will label subexpressions with unique flow variables. We will label expressions in two distinct ways, as a *source* of flow or as a *target* of flow. We will use e^a as our notation for labelling e (with flow variable a) as a source of flow and e_a as our notation for labelling e as a target of flow. If we are interested in the flow of values from *producers* to *consumers* then we label all program points where values are created as sources of flow, we label the points where values are

destructed as targets of flow, and we leave all other subexpressions unlabelled. In the example below we have labelled all values as sources with flow variables a_0 through a_4 and we have labelled the arguments to plus as targets with a_5 and a_6. We have not labelled the others consumers (the applications) to keep the example less cluttered.

$$\begin{array}{l}\texttt{let}\ apply = (\lambda f.(\lambda y.f\ y)^{a_0})^{a_1} \\ \texttt{in}\ \texttt{let}\ id = (\lambda x.x)^{a_2} \\ \quad \texttt{in}\ (apply\ id\ 5^{a_3})_{a_5} + (apply\ id\ 7^{a_4})_{a_6}\end{array}$$

We may now ask the question "which values may show up as arguments to plus?". Our flow analysis will give the answer that the value labelled with a_3 (5) may flow to a_5 (the first argument) and a_4 (7) may flow to a_6 (the second argument). In this example the flow polymorphic types that we assign to id and $apply$ plays a crucial role. A monomorphic system would conservatively say that both values could flow to both places. For some applications we might be interested in, not only the flow from producers to consumers, but also the flow to points on the way from a consumer to a producer. In our example we might be interested in the flow to x in the body of id. We then add a target label on x as in

$$\begin{array}{l}\texttt{let}\ apply = (\lambda f.(\lambda y.f\ y)^{a_0})^{a_1} \\ \texttt{in}\ \texttt{let}\ id = (\lambda x.x_{a_7})^{a_2} \\ \quad \texttt{in}\ (apply\ id\ 5^{a_3})_{a_5} + (apply\ id\ 7^{a_4})_{a_6}\end{array}$$

and then ask for the flow to a_7. Our analysis would answer with a_3 and a_4. An important property of the analysis is that the type of id remains polymorphic even though we tap off the flow passing through x. Thus our type system corresponds to the *sticky interpretation* of a type derivation in [Mos97]. The key to this property is to distinguish between source labels and target labels. If the label on x would serve as both a source and a target label the flow through id would be monomorphic. [3]

The language we consider is a lambda calculus extended with recursive let-expressions, integers, lists and case-expressions. The grammar of the language is as follows.

Variables	x, y, z		
Flow Variables	a		
Expressions	e	$::=$	$\lambda x.e \mid n \mid \texttt{nil} \mid \texttt{cons}\ e_0\ e_1 \mid x \mid e_0 + e_1 \mid e_0\ e_1 \mid$ $\texttt{let}\ \{\vec{b}\}\ \texttt{in}\ e \mid \texttt{case}\ e\ \texttt{of}\ alts \mid e^a \mid e_a$
Bindings	b	$::=$	$x = e$
Alternatives	$alts$	$::=$	$\{\texttt{nil} \Rightarrow e_0, \texttt{cons}\ x\ y \Rightarrow e_1\}$

The language is simply typed and for our complexity result we assume that the terms are explicitly typed by having type annotations attached to every

[3] We can achieve this degrading effect by annotating x both as a source and as a target but using the same flow variable, i.e., as $x_{a_7}^{a_7}$.

$$\frac{}{\top \vdash \mathtt{Int} \leq \mathtt{Int}} \qquad \frac{M \vdash \tau \leq \tau'}{M \wedge (a \leq a') \vdash (\mathtt{List}\,\tau)^a \leq (\mathtt{List}\,\tau')^{a'}}$$

$$\frac{M \vdash \tau_0' \leq \tau_0 \qquad N \vdash \tau_1 \leq \tau_1'}{M \wedge N \vdash \tau_0 \to \tau_1 \leq \tau_0' \to \tau_1'} \qquad \frac{M \vdash \rho_0 \leq \rho_1}{M \wedge (a \leq a') \vdash \rho_0^a \leq \rho_1^{a'}}$$

Figure 2: Subtyping rules

subterm. For our flow analysis we label the types of the underlying type system with flow variables.

$$\text{Flow Types} \quad \tau \quad ::= \quad \mathtt{Int}^a \mid (\tau \to \tau')^a \mid (\mathtt{List}\,\tau)^a$$

We will let ρ range over flow types without the outermost annotation. The subtype entailment relation which take the form $M \vdash \tau_0 \leq \tau_1$ is defined in Figure 2. Recall that M ranges over constraint terms as defined in Section 2.1. We read $M \vdash \tau_0 \leq \tau_1$ as "from the constraint term M it can be derived that $\tau_0 \leq \tau_1$". We will let σ range over type schemas.

$$\text{Type Schemas} \quad \sigma \quad ::= \quad \forall \vec{a}.\, f\,\vec{a} \Rightarrow \tau$$

Since the underlying type system is monomorphic type schemas will only quantify over flow variables. A type schema contains a call $f\,\vec{a}$ to a constraint abstraction which may constrain the quantified variables. We will let Θ and Δ range over typing contexts which associates variables with types or type schemas depending on whether it is a let-bound variable or not. We will use juxtaposition as our notation for combining typing contexts. Our typing judgements take the form $\Theta; M \vdash e : \tau$ for terms, $\Theta; F \vdash b : (x : \sigma)$ for bindings and $\Theta; \Gamma \vdash \{\vec{b}\} : \Delta$ for groups of bindings. (Recall that F ranges over constraint abstractions and that Γ ranges over sets of constraint abstractions.) The typing rules of the analysis can be seen in Figure 3 and 4. The key difference to the type system in [Mos97] is in the rule Binding where generalisation takes place. Instead of putting the constraint term used to type the body of the binding into the type schema the constraint term is inserted into a new constraint abstraction and a call to this abstraction is included in the type schema.

To compute the flow in a program we can proceed as follows. First we compute a principal typing of the program which includes a constraint term where the free variables are the flow variables labelling the program. We then apply the algorithm from Section 3 and extract a set of atomic constraints which we can view as a graph. If there is a path from a_0 to a_1 then a_0 may flow to a_1. The typing rules as presented here are not syntax directed and cannot directly be interpreted as describing an algorithm for computing principal typings.

$$\text{Abs}\frac{\Theta\{x:\tau\}; M \vdash e : \tau'}{\Theta; M \vdash \lambda x.e : (\tau \to \tau')^a} \quad \text{Int}\frac{}{\Theta; \top \vdash n : \texttt{Int}^a} \quad \text{Nil}\frac{}{\Theta; \top \vdash \texttt{nil} : (\texttt{List}\,\tau)^a}$$

$$\text{Cons}\frac{\Theta; M \vdash e_0 : \tau \qquad \Theta; N \vdash e_1 : (\texttt{List}\,\tau)^a}{\Theta; M \wedge N \vdash \texttt{cons}\,e_0\,e_1 : (\texttt{List}\,\tau)^a}$$

$$\text{Var-}\sigma\frac{}{\Theta\{x : \forall\vec{a}.f\;\vec{a} \Rightarrow \tau\}; f\;\vec{b} \vdash x : \tau[\vec{a} := \vec{b}]} \quad \text{Var-}\tau\frac{}{\Theta\{x:\tau\}; \top \vdash x : \tau}$$

$$\text{Plus}\frac{\Theta; M \vdash e_0 : \texttt{Int}^{a_0} \qquad \Theta; N \vdash e_1 : \texttt{Int}^{a_1}}{\Theta; M \wedge N \vdash e_0 + e_1 : \texttt{Int}^a}$$

$$\text{App}\frac{\Theta; M \vdash e_0 : (\tau \to \tau')^a \qquad \Theta; N \vdash e_1 : \tau}{\Theta; M \wedge N \vdash e_0\,e_1 : \tau'}$$

$$\text{Let}\frac{\Theta\Delta; \Gamma \vdash \{\vec{b}\} : \Delta \qquad \Theta\Delta; M \vdash e : \tau}{\Theta; \texttt{let}\,\Gamma\,\texttt{in}\,M \vdash \texttt{let}\,\{\vec{b}\}\,\texttt{in}\,e : \tau}$$

$$\text{Case}\frac{\Theta; M \vdash e : \tau \qquad \Theta; N \vdash alts : \tau \Rightarrow \tau'}{\Theta; M \wedge N \vdash \texttt{case}\,e\,\texttt{of}\,alts : \tau'}$$

$$\text{Alts}\frac{\Theta; M \vdash e_0 : \tau' \qquad \Theta\{x:\tau, y:(\texttt{List}\,\tau)^a\}; N \vdash e_1 : \tau'}{\Theta; M \wedge N \vdash \{\texttt{nil} \Rightarrow e_0; \texttt{cons}\,x\,y \Rightarrow e_1\} : (\texttt{List}\,\tau)^a \Rightarrow \tau'}$$

$$\text{Source}\frac{\Theta; M \vdash e : \rho^a}{\Theta; M \wedge (a \le c) \wedge (b \le c) \vdash e^b : \rho^c}$$

$$\text{Target}\frac{\Theta; M \vdash e : \rho^a}{\Theta; M \wedge (a \le c) \wedge (a \le b) \vdash e_b : \rho^c}$$

$$\text{Sub}\frac{\Theta; M \vdash e : \tau}{\Theta; M \wedge N \vdash e : \tau'}\; N \vdash \tau \le \tau'$$

$$\text{Exist-intro}\frac{\Theta; M \vdash e : \tau}{\Theta; \exists\vec{a}.M \vdash e : \tau}\; \{\vec{a}\} \cap \text{FV}(\Theta, e, \tau) = \emptyset$$

Figure 3: Typing rules for a flow analysis

$$\text{Binding group-}\emptyset\ \frac{}{\Theta; \emptyset \vdash \emptyset : \emptyset}$$

$$\text{Binding group}\ \frac{\Theta; \Gamma \vdash \{\vec{b}\} : \Delta \qquad \Theta; F \vdash b : (x : \sigma)}{\Theta; \Gamma\{F\} \vdash \{\vec{b}, b\} : (\Delta, x : \sigma)}$$

$$\text{Binding}\ \frac{\Theta; M \vdash e : \tau}{\Theta; f\ \vec{a} = M \vdash x = e : (x : \forall \vec{a}.f\ \vec{a} \Rightarrow \tau)}\ \{\vec{a}\} \cap \mathrm{FV}(\Theta, f\ \vec{a} = M, e) = \emptyset$$

Figure 4: Typing rules for a flow analysis

Firstly, the subsumption rule (Sub) and the rule (Exist-intro) which introduces existential quantification in constraints can be applied everywhere in a typing derivation. This problem is solved by the standard approach to incorporate (Sub) and (Exists-intro) into an appropriate subset of the other rules to obtain a syntax-directed set of rules. Secondly, in the rule (Let) an inference algorithm would have to come up with an appropriate Δ. However, this only amounts to coming up with fresh names: Clearly, Δ would have to contain one type associations of the form $x : \sigma$ for each variable defined by the let-expression. Recall that σ is of the form $\forall \vec{a}.f\ \vec{a} \Rightarrow \tau$. We obtain τ simply by annotating the underlying type with fresh flow variables. Since they are fresh we will be able to generalise over all of them so we can take $\vec{a}$ to be these variables in some order. Finally we generate the fresh name f for the constraint abstraction. Note that no fixed-point calculation is required which is possible because we have recursive constraint abstractions. Now let us apply the algorithm to our example program. We first compute the constraint term in the principal typing which yields the following.

$$\begin{aligned}
&\texttt{let}\ f_{apply}\ b_0\ b_1\ b_2\ b_3\ b_4\ b_5\ b_6 = \exists c_0.\exists c_1.\exists c_2.(b_3 \leq b_0) \wedge (b_1 \leq b_4) \wedge (b_2 \leq c_2) \wedge \\
&\qquad\qquad (c_1 \leq b_5) \wedge (a_0 \leq b_5) \wedge (c_0 \leq b_6) \wedge (a_1 \leq b_6) \\
&\texttt{in let}\ f_{id}\ b_0\ b_1\ b_2 = \exists c_0.\exists c_1.(b_0 \leq c_1) \wedge (c_1 \leq b_1) \wedge (c_1 \leq a_7) \wedge \\
&\qquad\qquad (c_0 \leq b_2) \wedge (a_2 \leq b_2) \\
&\quad \texttt{in}\ \exists c_0. \ldots \exists c_{18}.(f_{apply}\ c_0\ c_1\ c_2\ c_3\ c_4\ c_5\ c_6) \wedge (f_{id}\ c_0\ c_1\ c_2) \wedge \\
&\qquad\qquad (c_7 \leq c_3) \wedge (a_3 \leq c_3) \wedge (c_4 \leq c_8) \wedge (c_4 \leq a_5) \wedge \\
&\qquad\qquad (f_{apply}\ c_{10}\ c_{11}\ c_{12}\ c_{13}\ c_{14}\ c_{15}\ c_{16}) \wedge (f_{id}\ c_{10}\ c_{11}\ c_{12}) \wedge \\
&\qquad\qquad (c_{17} \leq c_{13}) \wedge (a_3 \leq c_{13}) \wedge (c_{14} \leq c_{18}) \wedge (c_{14} \leq a_5)
\end{aligned}$$

Then we apply the algorithm from Section 3 and extract the set of free live atomic constraints which is $\{a_3 \leq a_5,\ a_4 \leq a_6,\ a_3 \leq a_7,\ a_4 \leq a_7\}$. The paths in this constraint set (viewed as a graph) is the result of the analysis.

Finally, by inspecting the rules we can see that the size of the skeleton of the constraint term required to type a program is proportional to the size of the

explicitly typed program and that the number of free variables is the number of flow variables in the program. From this fact and theorem 3.12 we can conclude that the complexity of the flow analysis is $O(n^3)$ where n is the size of the typed program.

B Proof of Theorem 3.6

In this appendix we give a proof of Theorem 3.6. We first introduce a form of constraint term contexts, reminiscent of evaluation contexts used in operational semantics, where the hole may not occur under any binder.

$$\text{Evaluation Contexts} \quad E ::= \quad [\cdot] \mid E \wedge M \mid M \wedge E$$

Note that the hole in an evaluation context is always live. We have the following properties for evaluation contexts which we state without proof.

Lemma B.1

1. $\mathtt{let}\,\Gamma\,\mathtt{in}\,E[\mathtt{let}\,\Gamma'\,\mathtt{in}\,M]$ is in normal form iff $\mathtt{let}\,\Gamma\Gamma'\,\mathtt{in}\,E[M]$ is in normal form.

2. $\mathrm{LIVE}(\mathtt{let}\,\Gamma\,\mathtt{in}\,E[\mathtt{let}\,\Gamma'\,\mathtt{in}\,M]) = \mathrm{LIVE}(\mathtt{let}\,\Gamma\Gamma'\,\mathtt{in}\,E[M])$.

3. If $a \notin \mathrm{FV}(\Gamma, E)$, and $\mathtt{let}\Gamma\mathtt{in}E[\exists a.M]$ is in normal form then $\mathtt{let}\Gamma\mathtt{in}E[M]$ is in normal form.

The key to the proof of Theorem 3.6 is the following auxiliary relation.

Definition B.2
We define an auxiliary relation $\theta; \Gamma \bullet\models M$ as:
$\theta; \Gamma \bullet\models M$ iff there exists E such that

1. $\mathtt{let}\,\Gamma\,\mathtt{in}\,E[M]$ is in normal form,

2. $\theta \models \mathrm{LIVE}(\mathtt{let}\,\Gamma\,\mathtt{in}\,E[M])$,

3. $\mathrm{FAV}(\mathtt{let}\,\Gamma\,\mathtt{in}\,E[M]) = \emptyset$.

The technical core of the proof now shows up in the proof of the following lemma.

Lemma B.3
if $\theta; \Gamma \bullet\models M$ then $\theta; \Gamma \models M$.

Before we proceed with the proof of this lemma we will use it to establish Theorem 3.6.

Proof B.4 (Theorem 3.6)
Assume the premise. The right way implication (if $\theta; \emptyset \models M$ then $\theta \models \mathrm{LIVE}(M)$) follows the fact that all syntactically live constraints are semantically live. To show the left way implication (if $\theta \models \mathrm{LIVE}(M)$ then $\theta; \emptyset \models M$) assume that $\theta \models \mathrm{LIVE}(M)$ which immediately gives $\theta; \emptyset \bullet\models M$. Thus, by Lemma B.3, $\theta; \emptyset \models M$ as required.

Finally we prove Lemma B.3.

Proof B.5 (Lemma B.3)
Recall that $\theta; \Gamma \models M$ is defined coinductively by the rules in Figure 1. That is, $\models$ is defined as the largest fixed point of the functional $\mathcal{F}$ expressed by the rules. By the coinduction principle we can show that $\bullet\models \subseteq \models$ if we can show that $\bullet\models \subseteq \mathcal{F}(\bullet\models)$. Thus we assume that $\theta; \Gamma \bullet\models M$ and proceed by case analysis on the structure of M.

case $M \equiv a \leq b$**:** By the definition of $\theta; \Gamma \bullet\models a \leq b$ there exists E which fulfils the requirements in Definition B.2. In particular, $\theta \models \mathrm{LIVE}(\mathtt{let}\,\Gamma\,\mathtt{in}\,E[a \leq b])$. Since E cannot capture variables and the hole in E is live we know that $a \leq b \in \mathrm{LIVE}(\mathtt{let}\,\Gamma\,\mathtt{in}\,E[M])$ so $\theta; \Gamma\ \mathcal{F}(\bullet\models)\ a \leq b$.

case $M \equiv \top$**:** Trivial.

case $M \equiv K \wedge L$**:** To show that $\theta; \Gamma\ \mathcal{F}(\bullet\models)\ K \wedge L$ we need to show that $\theta; \Gamma \bullet\models K$ and $\theta; \Gamma \bullet\models L$. We will only show the former, the latter follows symmetrically. By the definition of $\theta; \Gamma \bullet\models K \wedge L$ there exists E which fulfils the requirements in Definition B.2. Take E' to be $E[[\cdot] \wedge L]$. Then E' is a witness of $\theta; \Gamma \bullet\models K$.

case $M \equiv \mathtt{let}\Gamma'\mathtt{in}N$**:** We may without loss of generality (due to properties of α-conversion) assume that the constraint abstraction variables defined Γ and Γ' are disjoint. To show that $\theta; \Gamma\ \mathcal{F}(\bullet\models)\ \mathtt{let}\Gamma'\mathtt{in}N$ we need to show that $\theta; \Gamma\Gamma' \bullet\models N$. By the definition of $\theta; \Gamma \bullet\models \mathtt{let}\ \Gamma'\ \mathtt{in}\ M$ there exists E which fulfils the requirements in Definition B.2. Floating of let bindings preserves normal forms (Lemma B.1) so we can float out Γ' and obtain $\mathtt{let}\Gamma\Gamma'\,\mathtt{in}\,E[M]$ in normal form. Also, by Lemma B.1, $\mathrm{LIVE}(\mathtt{let}\,\Gamma\,\mathtt{in}\,E[\mathtt{let}\,\Gamma'\,\mathtt{in}\,M]) = \mathrm{LIVE}(\mathtt{let}\,\Gamma\Gamma'\,\mathtt{in}\,E[M])$. Thus E is a witness of $\theta; \Gamma\Gamma' \bullet\models M$.

case $M \equiv f\,\vec{b}$**:** By the definition of $\theta; \Gamma \bullet\models f\,\vec{b}$ we know that f must bound by Γ, i.e., $\Gamma = \Gamma'\{f\,\vec{a} = N\}$ for some Γ' and some N. We are required to show that $\theta; \Gamma'\{f\,\vec{a} = N\} \bullet\models N[\vec{a} := \vec{b}]$. From $\theta; \Gamma \bullet\models f\,\vec{b}$ we know that there exists E which fulfils the requirements in Definition B.2. Normal forms are closed under

unwindings (Lemma 3.7) so $\texttt{let}\,\Gamma'\{f\,\vec{a} = N\}\,\texttt{in}\,E[N[\vec{a} := \vec{b}]]$ is in normal form. Also, by Lemma 3.8,

$$\text{LIVE}(\texttt{let}\,\Gamma'\{f\,\vec{a} = N\}\,\texttt{in}\,E[f\,\vec{b}]) = \text{LIVE}(\texttt{let}\,\Gamma'\{f\,\vec{a} = N\}\,\texttt{in}\,E[N[\vec{a} := \vec{b}]]).$$

Thus E is a witness of $\theta; \Gamma'\{f\,\vec{a} = N\} \bullet\!\models N[\vec{a} := \vec{b}]$.

case $M \equiv \exists a.N$ To show that $\theta; \Gamma\ \mathcal{F}(\bullet\!\models)\ \exists a.N$ we need to show that there exists $d \in \mathcal{L}$ such that $\theta[a := d]; \Gamma \bullet\!\models N$. Let

$$d = \bigsqcup\{\theta(a') \mid a' \neq a \text{ and } a' \leq a \in \text{LIVE}(N)\}.$$

By the definition of $\theta; \Gamma \bullet\!\models \exists a.N$ there exists E which fulfils the requirements in Definition B.2. Without loss of generality (due to properties of α-conversion) we can assume that $a \notin \text{FV}(\Gamma, E)$. Since $\texttt{let}\,\Gamma\,\texttt{in}\,E[\exists a.N]$ is in normal form, and $a \notin \text{FV}(\Gamma, E)$ we know, by Lemma B.1, that $\texttt{let}\,\Gamma\,\texttt{in}\,E[N]$ is in normal form. It remains to show that $\theta[a := d] \models \text{LIVE}(\texttt{let}\,\Gamma\,\texttt{in}\,E[N])$. Given $A \in \text{LIVE}(\texttt{let}\,\Gamma\,\texttt{in}\,E[N])$ we proceed by the following cases.

subcase $A \equiv a \leq a$**:** Trivial.

subcase $A \equiv b \leq c$ **where** $b \neq a$ **and** $c \neq a$**:**
In this case $A \in \text{LIVE}(\texttt{let}\,\Gamma\,\texttt{in}\,E[\exists a.N])$ so $\theta \models A$ and thus $\theta[a := d] \models A$.

subcase $A \equiv b \leq a$ **where** $b \neq a$ **:** In this case $A \in \text{LIVE}(N)$ and thus $\theta[a := d] \models A$ by the construction of d.

subcase $A \equiv a \leq b$ **and** $b \neq a$**:** In this case $a \leq b \in \text{LIVE}(N)$. We will show that $\theta(b)$ is an upper bound of

$$\{\theta(a') \mid a' \neq a \text{ and } a' \leq a \in \text{LIVE}(N)\}$$

and, since d is defined as the least upper upper bound, $\theta[a := d] \models a \leq b$ follows. Now given any a' such that $a' \neq a$ and $a' \leq a \in \text{LIVE}(N)$. Since $a' \leq a \in \text{LIVE}(N)$ and $a \leq b \in \text{LIVE}(N)$ we know that $\texttt{let}\,\Gamma\,\texttt{in}\,E[\exists a.N] \rightarrow \texttt{let}\,\Gamma\,\texttt{in}\,E[\exists a.N \wedge a' \leq b]$ and since $\texttt{let}\,\Gamma\,\texttt{in}\,E[\exists a.N]$ is in normal form we know that

$$\text{LIVE}(\texttt{let}\,\Gamma\,\texttt{in}\,E[\exists a.N \wedge a' \leq b]) = \text{LIVE}(\texttt{let}\,\Gamma\,\texttt{in}\,E[\exists a.N]).$$

Finally, since $a' \neq a$ it must be the case that $a' \leq b \in \text{LIVE}(\texttt{let}\,\Gamma\,\texttt{in}\,E[\exists a.N \wedge a' \leq b])$ and thus $a' \leq b \in \text{LIVE}(\texttt{let}\,\Gamma\,\texttt{in}\,E[\exists a.N])$. Hence $\theta \models a' \leq b$ so $\theta(a') \sqsubseteq \theta(b)$.