UNIVERSITY OF GOTHENBURG

Utilizing the Value State Dependence Graph for Haskell

Master of Science Thesis in Computer Science

Nico Reißmann

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, May 2012

Utilizing the Value State Dependence Graph for Haskell

Nico Reißmann

Examiner: Björn von Sydow

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden May 2012

Abstract

Modern compilers use control flow based intermediate representations for representing programs during code optimization and generation. However, many optimizations tend to rely not on the explicit representation of control paths, but on the flow of data between operations. One such intermediate representation that makes this flow explicit is the Value State Dependence Graph (VSDG). It abandoned explicit control flow and only models the data flow between operations. The flow of control is at a later point recovered from these data flow properties.

The goal of this thesis is to make the Value State Dependence Graph applicable for Haskell. This is accomplished by equipping the GHC compiler with a proof-of-concept back-end that facilitates the use of it. The new back-end allows for a simplified compilation process by translating a program at an early stage into a VSDG and expressing all further transformations down to assembly code in it. Also, the back-end is theoretically evaluated and a comparison is drawn between it and the already present ones.

Acknowledgements

Firstly, I would like to thank my supervisor, Prof. Dr. Björn von Sydow, for his unending enthusiasm, encouragement and invaluable guidance throughout the course of this project. I would also like to thank my friend Dr. Helge Bahmann for his support, feedback and advice. My parents deserve countless thanks for their support and understanding throughout my time of study. Without these people this thesis would have never been possible. Finally, special thanks go to Giorgia Dallera for the beautiful image on the cover page.

Contents

1 Introduction

Modern compilers are among the largest and most complex software systems in existence. In order to keep this complexity in check, a compiler is usually implemented through several stages, where each stage can at least conceptually be assigned to one of the two components: the *front-end* and the *back-end*. Ideally, the front-end is only concerned with language specific details, containing stages for parsing and type-checking of the language, while the back-end takes care of the underlying hardware details, containing stages for code optimization and the production of assembly or machine code. Both components share a common intermediate representation (IR) serving as glue between them as shown in figure 1.1.

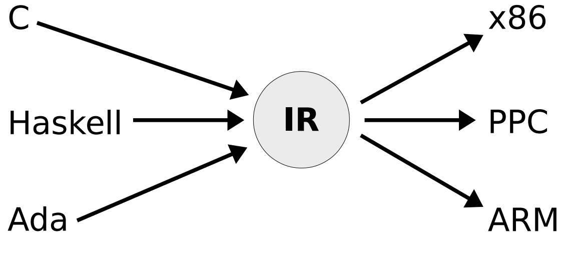

Figure 1.1: *Compiler for several languages and target machines*

Since the back-end is relatively independent of the source language, it is possible to share it with its individual stages like code optimization, instruction selection and register allocation among different compilers. This fact gave rise to a great amount of research in the design and implementation of back-ends, offering several approaches to compiler writers nowadays besides the obvious generation of assembly or machine code.

One such an approach is to compile the source language into another high level language and let the compiler for this language do the rest of the work. A language often used as such a target language is C [7, 10, 35, 11]. While this approach clearly benefits from the availability of high quality C compilers on most platforms, it also has a number of disadvantages. Firstly, C was never meant as the target language of a compiler. It lacks certain features of modern programming languages such as higher-order functions or garbage collection which makes it hard to map certain source languages to it. A great amount of work is often necessary in order to accomplish this mapping, mostly leading to C code that is difficult to optimize for the compiler. A second disadvantage is clearly the loss of control over the code generation stages.

Another approach is to target a high level assembly language. These languages are a compromise between a high level language and normal assembly, offering more control over code generation while still being abstract enough to be platform independent. One famous representative is the Low Level Virtual Machine (LLVM) [1]. It is a compiler infrastructure centered on such a language, providing a great variety of code optimizations.

All three possibilities, i.e. native assembly, C and the LLVM language, exist as an option for code generation in the Glasgow Haskell Compiler (GHC) [2], a cutting-edge compiler for the Haskell programming language [28]. The introduction of the LLVM back-end in 2009 enabled GHC to use the many optimizations accompanied in the LLVM compiler suite. However, LLVM uses still a control-flow based representation as intermediate language and therefore, as base for the many optimizations that are necessary for producing efficient machine or assembly code. While it is easy to construct a control-flow graph and produce machine/assembly code from it, it is generally tedious and complicated to perform certain kind of optimizations on it, including even the necessity of constructing other IRs, e.g. control-dependence graph, in order to produce efficient code. Furthermore, common and necessary optimizations such as common subexpression elimination, loop/conditional invariant code motion, dead code elimination etc. are based on the flow of data and not the flow of control.

In 2003, Neil Johnson and Alan Mycroft introduced a new data flow based intermediate representation called the Value State Dependence Graph (VSDG). The VSDG is a directed graph consisting of nodes that represent operations, conditionals and loops. Furthermore, two kind of edges for representing value- and state-dependencies are present. The value-dependency edges indicate the flow of values between the nodes in the graph, whereas the state-dependency edges introduce sequential ordering into the graph. In his PhD thesis [20], Alan C. Lawrence argues for the use of the VSDG instead of a traditional control flow based representation, since it is generally significantly cheaper to perform optimizations on this data flow based representation.

The overall goal of the author of this thesis is to equip the Glasgow Haskell Compiler with a VSDG back-end. This is done with the help of Jive [3], a compiler back-end using

[1]http://llvm.org

[2]http://www.haskell.org/ghc/

[3]http://www.jive-compiler.chaoticmind.net

the Value State Dependence Graph as its intermediate representation. However, since Haskell is a huge language and Jive is just in a beta stadium, it would be unfeasible to fully support the entire Haskell language in a half year project like this. Therefore, the goal of this thesis is to give a proof-of-concept by limiting the implementation to only support the signed fixed-point integers of Haskell. Of course, all the known features such as higher-order functions, polymorphism and lazy evaluation etc. will be present in the implementation, but just on the basis of signed fixed-point integers.

1.1 Organization of this Thesis

In order to highlight the motivation for this thesis, chapter 2 discusses the current state of the Glasgow Haskell Compiler. It also describes all the necessary internals of GHC which are needed to understand the rest of this thesis. Chapter 3 explains the fundamentals of the Jive compiler back-end together with all operations supported by Jive. Building on the gained knowledge of the two previous chapters, chapter 4 presents the implementation of the new back-end for the Glasgow Haskell Compiler. Finally, chapter 5 evaluates the new back-end and discusses the future work for the back-end and Jive itself.

2 The Glasgow Haskell Compiler

The Glasgow Haskell Compiler is a modern and heavily optimizing compiler for the Haskell programming language. Since its first beta release in 1991, it has become the de facto standard compiler for Haskell. This chapter gives an overview of the compilation process for Haskell by examining the compilation pipeline of GHC. It especially focuses on the back-end of the compiler with its different possibilities for code generation. The benefits and drawbacks along with the limitations of each possibility are pointed out.

Furthermore, a closer look is taken at the core language, an intermediate representation that was especially designed to express the full range of Haskell, but not more. Basically, it is an enriched form of the lambda calculus. The core language is later on used as the starting point for the compilation with the Jive compiler.

Last, but not least, primitive types and operations are further examined. These primitives build the raft of the Haskell programming language. Since they cannot be expressed in Haskell and the Haskell code generator is circumvented, they must be mapped to operations in the Jive compiler.

2.1 The Compilation Pipeline

In general, compilers tend to be split into individual stages, where each stage is devoted to exactly one task of the compilation process. This greatly reduces the complexity of the overall system. Each stage takes an intermediate representation of the program as input and outputs an altered or another representation of the program. Figure 2.1 shows a simplified overview of the compilation pipeline of the Glasgow Haskell Compiler [15] with its most important stages.

A Haskell module is first parsed and type-checked using the HSSyn representation. This IR represents Haskell in its full-blown syntax. During type-checking HSSyn is further annotated with types that were inferred from the program using the Hindley-Milner [23] type inference algorithm. Since type-checking is done on the HSSyn which represents a Haskell program with all its syntax, the generation of error messages is greatly eased. After type-checking, the verbose HSSyn representation is desugared [29] into the core language. Basically, the core language is System F [30] extended with type equality coercions [34]. It was especially designed to be large enough to express the full range of Haskell, but not larger. It is further explained in section 2.2.

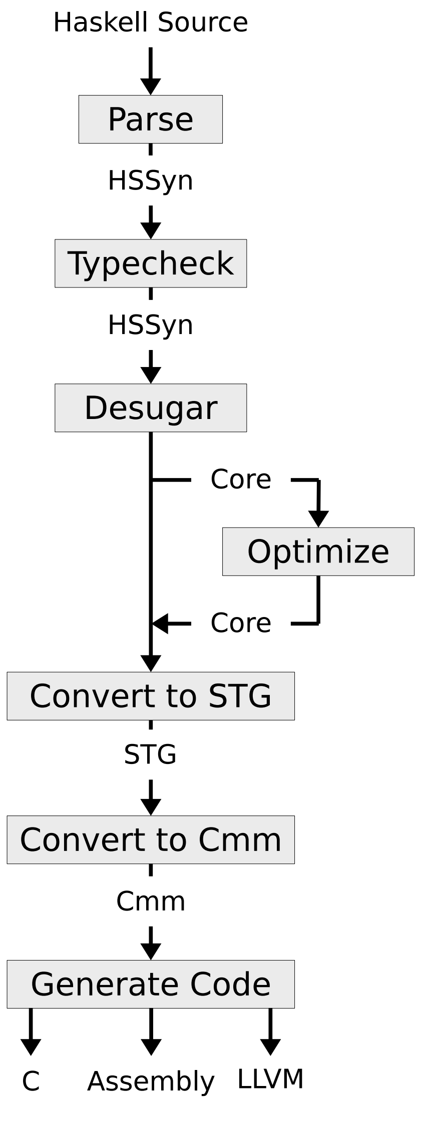

Figure 2.1: *The GHC Compilation Pipeline*

After several optimizations on the core language, which are further explained in section 2.1.1, a translation into the Spineless Tagless G-Machine language [14] takes place. This IR is still functional with the usual denotational semantics, however, it has also a well defined operational semantics attached to it. It is further explained in section 2.1.2. After narrowing the gap between the functional world and stock hardware with the help of the STG machine, a variant of C–– [18] is produced as output. C–– is a portable assembly language and was especially designed to be used as the target of a compiler. It serves as a common starting point for the different code generators. At the time of writing of this thesis GHC supports three different ways of producing target machine code. Firstly, it can just pretty print the C–– language and let a C compiler handle the rest. Secondly, it can directly produce the native machine code for the underlying hardware. Finally, it is possible to output LLVM virtual machine code for the compilation with the LLVM compiler suite [36]. The different approaches are further explored in section 2.1.3.

2.1.1 Optimizer

The optimizer of GHC works entirely on the Core language, performing only Core-to-Core transformations in order to improve program performance. The transformations of the optimizer can be divided into two groups. The first group is a large set of simple and local transformations such as inlining [27], constant folding or beta reduction. These transformations are all implemented in one single and complex pass which is called the *simplifier*. The second group is a set of complex and global transformations such as strictness analysis [16] or specializing overloaded functions. The majority of the transformations in this group consist of an analysis phase, followed by a transformation phase that uses the results of the analysis. Since certain optimizations expose opportunities for other optimizations, the simplifier is applied repeatedly until no further changes occur or a certain number of iterations is reached. Further optimizations performed are eta expansion, deforestation [38], constructed product result analysis [4] and let-floating [17]. A comprehensive list of the transformations can be found in [21].

2.1.2 Conversion to STG

The Spineless Tagless G-Machine is an abstract machine that consists mainly of two parts: its own functional language, the STG language, and an operational semantics defined for it. The language itself is similar to the Core language which is discussed in section 2.2. The biggest difference is that types have mainly been discarded; only enough type information was retained for guiding code generation. Additionally, an STG program is also explicitly decorated with information from some analyses such as a list of free variables for lambda-forms. However, that information was always present in the program and is just extracted from it in order to further ease the generation of code.

The second part is the operational semantics attached to the STG language. It clearly defines how to execute a program in the STG language on stock hardware. Along with this semantics, the STG defines a number of virtual registers such as for the heap and stack, but also general purpose registers that are, for example, needed for argument passing to function calls. In order to achieve reasonable performance, most of these registers, especially the special purpose ones such as the stack and heap pointer registers, need to be pinned to real hardware registers. The compilation mode where this is done is called *registered* mode[1]. However, register pinning is of course highly hardware specific; for each new architecture to be supported, a new mapping from virtual registers to hardware registers is needed. Also, it is not possible to just change the mapping for an already supported architecture, since this would mean that code which was compiled with the old mapping cannot work together with code compiled with the new mapping. Furthermore, pinned registers are excluded from register allocation, which render them useless in code parts where their content is not used and it would have been helpful to have another register at hand. This is especially a setback on architectures with a sparse register set such as the *i386* architecture. Additionally, the pinning of registers increases the complexity of the following code generation phases, since they have to be aware of the pinned registers. In summary, the STG machine not just dictates the outcome of the code generation phases by defining how to translate the STG language, but also has a pervasive impact on the complexity of the following phases and the portability of the entire compiler itself.

2.1.3 Code Generation

The code generation phase of the compiler translates the STG language by using the operational semantics defined for it into Cmm which is a variant of the C−− language and therefore also a variant of the C programming language. This language serves as the common starting point for the different code generators. At the time of writing of this thesis, three different kind of code generation phases exist: one which outputs C code, another one which produces native assembly code and finally, a phase that outputs LLVM assembly. All three generators are explained in greater detail below.

Additionally, further optimizations are performed on the Cmm language using hoopl [25]. Hoopl is a Haskell library that can be used to implement program transformations based on data flow analyses. It is used in GHC to improve the Cmm output of the compiler and ease the code generation for the following phases. This is accomplished by applying control flow and data flow optimizations such as unreachable/common block elimination, dead code elimination, constant propagation etc [2].

C Code Generator

The C code generator (CCG) is in fact quite simple and could almost be considered to be just pretty-printing of the Cmm language. GHC relies on the GNU GCC C compiler

[1]There also exists an *unregistered* mode where all those registers are stored on the heap needing memory reads and writes in order to access them. Of course, this is incredibly slow and not further discussed here, since this mode is mainly used for portability reasons.

[2]See http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/IntegratedCodeGen

(GCC) [3] in order to bring the code into a machine executable format. This solution is quite portable, at least for *POSIX* platforms. However, as mentioned in section 2.1.2, in order to be able to produce reasonable code, register pinning needs to be performed and there is no way to express this in the C programming language. Hence, GHC relies on a GCC specific feature called *global register variables* [4] which enables it to fix C variables to specific hardware registers. Of course, by using such a feature from GCC, GHC gets tied to GCC. Ongoing work is then required to keep everything working with new versions of GCC. Furthermore, by invoking another compiler for producing target code, the speed of compilation is dramatically reduced.

Native Code Generator

The native code generator (NCG) is able to produce native machine code for different architectures. At the time of writing of this thesis, *x86*, *SPARC* and *PowerPC* are supported. Of course, with this kind of approach it is no problem to support register pinning, since GHC has full control over the target code generation process. However, this power is paid with the price of complexity. The native code generator back-end is in comparison to the C or the LLVM back-end quite complicated, since it needs to take care of all those things that are handled externally by the others, i.e. register allocation, instruction selection etc.. Furthermore, even more work needs to be put into this back-end in order to produce reasonable code, since GHC cannot rely on external tools to perform any kind of optimizations. Of course, since no external tool needs to be called, the speed of compilation is much faster than for the C back-end.

LLVM Code Generator

The LLVM back-end was added fairly recently to GHC and enables the compiler to produce LLVM assembly. The Cmm language is directly translated into the LLVM assembly language and the LLVM toolchain is invoked in order to produce native machine code. This approach has the same problem with the register pinning as the C back-end: there is no way to tell LLVM to pin variables to certain registers. However, the author of this back-end came around this problem by implementing a new calling convention for the LLVM assembly language. It turns out that this even gained some nice speed ups for some programs, since the register pinning of the CCG and the NCG is an overdoing and the virtual registers just need to be in the right hardware register before each function call and exit, meaning that those registers could be freely used in the other parts of the code. Furthermore, GHC gained quite a lot for certain kind of programs by the entire bag of optimizations LLVM comes along with. However, since the LLVM assembler is invoked externally like the C compiler for the C back-end, it suffers from quite a loss of speed in comparison to the NCG even though it is not as tremendous as for the C back-end.

[3] http://gcc.gnu.org

[4] http://gcc.gnu.org/onlinedocs/gcc/Explicit-Reg-Vars.html

2.2 The Core Language

After a Haskell module has been parsed and type-checked, it is desugared into the Core language. The Core language is a tiny language that is large enough to express the full range of Haskell, but not larger. Listing 2.1 shows the data types used for representing the Core language in the GHC compiler at the time of writing of this thesis.

```
data Var =
  ...
  | Id {
    varName :: !Name,
    varType :: Type,
    id_details :: IdDetails,
    ...
  }

type Id = Var

type CoreExpr = Expr Var
type CoreArg = Arg Var
type CoreAlt = Alt Var
type CoreBind = Bind Var

data Expr b
  = Var Id
  | Lit Literal
  | App (Expr b) (Arg b)
  | Lam b (Expr b)
  | Let (Bind b) (Expr b)
  | Case (Expr b) b Type [Alt b]
  | Cast (Expr b) Coercion
  | Note Note (Expr b)
  | Type Type
  | Coercion Coercion

type Arg b = Expr b

type Alt b = (AltCon, [b], Expr b)

data AltCon
  = DataAlt DataCon
  | LitAlt Literal
  | DEFAULT

data Bind b
  = NonRec b (Expr b)
  | Rec [(b, (Expr b))]
```

Listing 2.1: *Data types used for representing the Core language in GHC.*

As you can see in the listing, the *CoreBind*, *CoreAlt*, *CoreArg* and *CoreExpr* are just type synonyms with a binding of *Var* to the corresponding real data types. The only interesting part of the Var data type for this thesis is its *Id* alternative which is used for representing identifiers. Each Id has a name, type and additional information attached to it that gives further information about its kind. The following sections explain the parts of the Core language that are important for code generation with the Jive compiler. For further information about the Core language consult the GHC Developer Wiki[5].

Bindings

Essentially, a Haskell module after desugaring into Core is not more than a list of CoreBinds. Every *Bind* element in this list was either created with the NonRec or Rec data constructor. The NonRec data constructor is used for constants and non-recursive functions, whereas the Rec data constructor is used for mutual recursive functions. Basically, each binding is not more than an expression associated with an identifier.

Variables

The *Var* data constructor in the data declaration of *Expr* represents the occurrence of an identifier in a Core expression. As shown in the code listing, the argument given to the constructor is an *Id* which is only a type synonym for the data type Var. This data type is used among other things for creating identifiers with its Id alternative manifesting the kind of the Id in the *id_details* component.

Different kind of identifiers are supported at the time of writing:

- *Vanilla id*: A vanilla id is used for ordinary Haskell function calls. It has the arity of the function attached to it.
- *Data Constructor id*: This identifier is used for creating the individual alternatives of algebraic data types. It carries besides other information a unique tag (integer) with it that is associated with each data constructor of an algebraic data type definition and used during pattern matching for distinguishing them from one another.
- *Primitive Operation id*: A primitive operation identifier indicates the presence of a primitive operation[6]. The identifier has a particular primitive operation associated with it.

[5]http://hackage.haskell.org/trac/ghc/

[6]See section 2.3.

Literals

Literals are expressed in Core through the *Lit* data constructor. The literals supported by Core are closely related to the primitive data types underlying the Haskell implementation in GHC[7] and therefore, the *Literal* data type (not shown in the code listing) has constructors for characters, (un-)signed integers and floating point number literals.

Lambda Abstractions

The *Lam* data constructor is used for lambda abstractions. It takes a variable as first argument and an expression as second argument. Since the constructor only takes one variable, every multi-parameter function can just be expressed by a nested lambda expression.

Application

The *App* data constructor is used for applying an argument to an expression. For example, this is used for applying the arguments to a function, but also for handing the individual arguments to a data constructor. Additionally, the App constructor is also used for specializing the types of functions by applying a type expression to them. Note that the argument of the apply operator is not a list, but only a single expression. This coincides with the lambda abstractions in Core and the applications need therefore also to be nested in order to handle multi-parameter functions.

Let Bindings

Let bindings are supported with the *Let* data constructor. The constructor has two arguments: a binding and an expression. Basically, the semantics is that a new scope is introduced for the expression given as second argument to the constructor. The bindings (it could be more than one binding in the recursive alternative) given as first argument overshadow in this scope all equivalent named bindings that were earlier defined.

Case Expressions

The *Case* expression is used for pattern matching and evaluation of expressions, since it is strict in its scrutinee. Its general form is *Case scrt bndr ty alts*, where

- *scrt* is the scrutinee (i.e. the expression under evaluation) of the case
- *bndr* is the case binder binding the value of the scrutinee to this name for later use in any of the alternatives

[7]See section 2.3.

- *ty* is the type of the case expression and equivalent to the type of every single alternative
- *alts* are the alternatives of the case expression

As you can see in listing 2.1, an alternative is a triple consisting of a constructor, a list of variables and the right-hand side of this alternative. Three options of alternative constructors are possible:

- a data constructor indicated by the *DataAlt* alternative constructor. This is used for inspecting data types defined by the data keyword. Operationally, this corresponds to checking whether a data type was constructed by this constructor and binding the variables to the constructors arguments so that they can be used in the right-hand side of the alternative.
- a literal indicated by the *LitAlt* alternative constructor. This is used for checking individual values of literals.
- a default option indicated by the *DEFAULT* alternative constructor. For example, this constructor occurs when not all possibilities were tested in a pattern match or the wild card was used in the match. It is further used when an expression just needs to be evaluated, since a case is the only construct being strict in Core.

The list of alternatives is always exhaustive, either because all alternatives are listed or because the default option is present in the list.

Type Expressions

The *Type* expression is used for specializing polymorphic functions by initializing a function's type variables to the argument of the constructor. It only occurs in the argument expression of an application.

Cast Expressions

As the name implies, the *Cast* expression casts the type of the expression given as first argument to the type given as second argument. This is used for implementing newtypes and generalized algebraic data types (GADTs) [6, 39].

2.3 Primitive Types and Operations

At the heart of GHC lie a number of primitive data types and operations, which build the foundation for the Haskell programming language. They are primitive in the sense that they cannot be implemented in Haskell itself and therefore need to be provided externally. Of course, in order to keep complexity in check it is crucial to have an as small as possible number of primitives while still being able to deliver reasonable performance and all the functionality to the Haskell programming language. The next two sections give an overview of primitive types and operations, while an exhaustive documentation of all primitive types and operations can be found in the GHC library documentation.

2.3.1 Primitive Types

Primitive types are the raft of the implementation of the Haskell language in GHC. There is no level of indirection or abstraction associated with primitive types and the representation of their values in memory are the same as with their equivalent types in C. For example, the Haskell *Int* type is a compound type whose values are represented by a pointer to a heap object, while the values of the primitive *Int#*[8] type corresponds to a bare integer itself. However, this does not necessarily mean that the values of primitive types cannot be represented by pointers to heap-allocated objects. An example of such a case would be *Array#*, which is generally a too big value to fit into a register and is therefore allocated on the heap. Starting with primitive types as a basis, it is easily possible to implement the compound types as the example in listing 2.2 shows for the Int type.

```
data Int = I# Int#
```

Listing 2.2: *Implementation of the Haskell Int type.*

2.3.2 Primitive Operations

Along with primitive types come operations for working on them. These operations implement the basic features of Haskell. On the one hand, some of them are quite simple and can usually be implemented by only emitting a short sequence of code or even just one simple instruction. One such example would be (+#), the addition on primitive integers, which coincides with a simple addition instruction. Since the translation of such primitive operations is quite straightforward, the code generator handles them and emits code for them during program translation. Such primitive operations are called *inlined*. On the other hand, the *out-of-line* operations are implemented by hand-written C−− code and linked to the program at the end of the compilation process along with the run-time system. Examples for such operations would be *raise#* and *catch#*, which are used for raising and catching exceptions in Haskell. Since primitive operations are not part of Haskell and need to be provided externally, calls to these functions need always to be saturated. Starting with the primitive operations as a basis again, the implementation of an operation on a compound data type can be easily accomplished. An example for the compound Int type is shown in listing 2.3.

[8]The # symbol has no special meaning and is only used as a syntactically convention inside GHC to distinguish primitive data types and operations from non-primitive ones. In order to use such names, the *-XMagicHash* extension needs to be enabled.

```haskell
plusInt :: Int -> Int -> Int
plusInt (I# x) (I# y) = I# (x +# y)
```

Listing 2.3: *Addition implementation for the Int type.*

3 The Jive Compiler

For many years, compilers used control flow based intermediate representations such as the Control Flow Graph (CFG) [3] for representing programs during the optimization and code generation phases of the compilation process. While the CFG is still the IR of choice in many compilers such as GCC and LLVM [19], many optimizations tend to rely not on the explicit representation of control paths, but on the flow of data between operations. One such intermediate representation which makes this flow explicit is the Value State Dependence Graph. It is a graph-based IR which abandons explicit control flow and only models the flow of data between operations. The flow of control is at a later point recovered from data flow properties.

This chapter is split in two parts. The first part is devoted to the Value State Dependence Graph, giving a formal definition of it and in order to get a first feeling for the mapping between a programming language and the VSDG, two examples of graphs for simple programs of the C programming language are shown. The C programming language was chosen over Haskell, since it has a rather intuitive mapping to the VSDG and assembly language.

The second half of the chapter deals with the Jive compiler itself, which uses the Value State Dependence Graph as intermediate representation of choice. All operations that are necessary for the Jive back-end of chapter 4 are discussed in detail. Furthermore, the compilation process starting with a Value State Dependence Graph as input and assembly language as output is outlined.

3.1 The Value State Dependence Graph by Definition

According to [20], the Value State Dependence Graph is formally defined as follows:

Definition 3.1.1. A Value State Dependence Graph is a directed labeled hierarchical Graph G = $(T, S, E, l, S_0, S_\infty)$, where:

Transitions T are nodes that represent operations. These may be *complex*, i.e contain a distinct graph G'.

Places S are nodes that represent the *results* of operations.

Edges $E \subseteq S \times T \cup T \times S$ represent dependencies on and production of results by operations.

Labeling Function l associates with every transition an operation.

Arguments $S_0 \subset S$ indicate places that contain the arguments upon entry of a function.

Results $S_\infty \subset S$ indicate places that contain the results upon exit of a function.

Every place is of a specific *type*, either *value* or *state*. Like places, every edge is also of a specific type, either value or state. An edge's type is defined by its endpoints: it is a state/value edge if and only if its place endpoint is a state/value.

Transitions represent operations in the Value State Dependence Graph through a labeling function l associating an operator with them. An *input* I_T of a transition T is a place connected to it by an edge and is said to be consumed by the transition. The transition is said to be a *consumer* of this place. Similar, a place is called an *output* O_T of a transition T if there exists an edge from the transition to the place. The transition is said to be the *producer* of this place. The set of inputs of a transition T is called its *operands* or *inputs* IS_T and the set of outputs of T is called its *results* or *outputs* OS_T.

Well-Formedness Requirements

Several requirements are necessary for a Value State Dependence Graph to be well-formed:

- **Acyclicity:** There are supposed to be no graph-theoretic cycles in a VSDG.
- **Node arity:** Every place must have a unique producer (i.e. a distinct incoming edge $\in T \times S$). This is equivalent to Static Single Assignment (SSA) form [9] of other intermediate representations. An exception are argument places S_{in}, which must have no incoming edges.
- **States must be used linearly:** Every state that is *changed* must be consumed exactly once.

Since it is a necessity that $S \cap T = \emptyset$, rendering the graph a bipartite graph, it is possible to give a simplified definition similar to [12] of the Value State Dependence Graph in terms of definition 3.1.1.

Definition 3.1.2. A Value State Dependence Graph is a directed labeled hierarchical Graph G=(N, E, l, N_0, N_∞), where

Nodes $N = (IS_T, T, OS_T)$ represent operations with $IS_T, OS_T \in S$, $IS_N = IS_T$ denoting the inputs of a node and $OS_N = OS_T$ denoting the outputs of a node. Like in definition 3.1.1, a node can be complex, i.e. containing a distinct graph G'.

Edges $E = OS_{N_1} \times IS_{N_2}$ with $N_1, N_2 \subseteq N$ and $N_1 \neq N_2$ represent dependencies on results of operations. Edges have their type inherited from its places (which have always the same type in a well-formed VSDG).

Labeling Function l associates with every node an operation.

Entry Nodes $N_0 = (IS_T, T, OS_T)$ with $N_0 \subset N$, $IS_T = \emptyset$ and $OS_T = S_0$ indicate the entry of functions.

Exit Nodes $N_\infty = (IS_T, T, OS_T)$ with $N_\infty \subset N$, $OS_T = \emptyset$ and $IS_T = S_\infty$ indicate the exit of functions.

It is important to be able to distinguish between multiple operands of nodes, rendering IS_N and OS_N rather tuples than sets. Firstly, the *order* of operands may be important, since not all operations are commutative (e.g. $x - y \neq y - x$). Secondly, an edge may be connected to two inputs of a node (e.g. $x + x$). Hence the type of the operands or results of a node is a tuple of the types of each operand or result, respectively.

Nodes

The VSDG supports three different kind of nodes: *computations*, γ nodes and *complex* nodes. Computation nodes model simple low-level operations. They can be categorized further into:

- **Value nodes** have only values as inputs and outputs. A special value node is a constant node, which has no inputs. Value nodes represent operations without side effects such as addition or subtraction.
- **State nodes** have mixed inputs and/or outputs and represent stateful operations with side effects such as load or store.

The second kind are γ nodes. A γ node is used for expressing conditional behavior in the VSDG. It multiplexes between two tuples of inputs t and f on basis of an input p acting as predicate. Both tuples of operands must have the same type r which is also the result type of the γ node. Depending on the run-time value of the predicate, either the t set of operands or the f set is passed through. The γ node is the only node in the Value State Dependence Graph that expresses *non-strict* behavior.

The last type of nodes are complex nodes or *regions*. A region contains a distinct graph G' which can be substituted for it. The graph G' can itself contain regions and therefore, regions form a hierarchical structure. The boundaries of such a region are fluent, meaning that nodes can be moved under certain conditions to an outer or inner region. However, the nesting property also puts a necessary restriction on edges: value and state edges are only allowed to connect to a node residing in the same region or a child region. Thus, regions restrict the reference of values similar to lexical scoping in programming languages. As we will see in section 3.3.3 and 3.3.6, this property makes them important for optimizations and sequentialization.

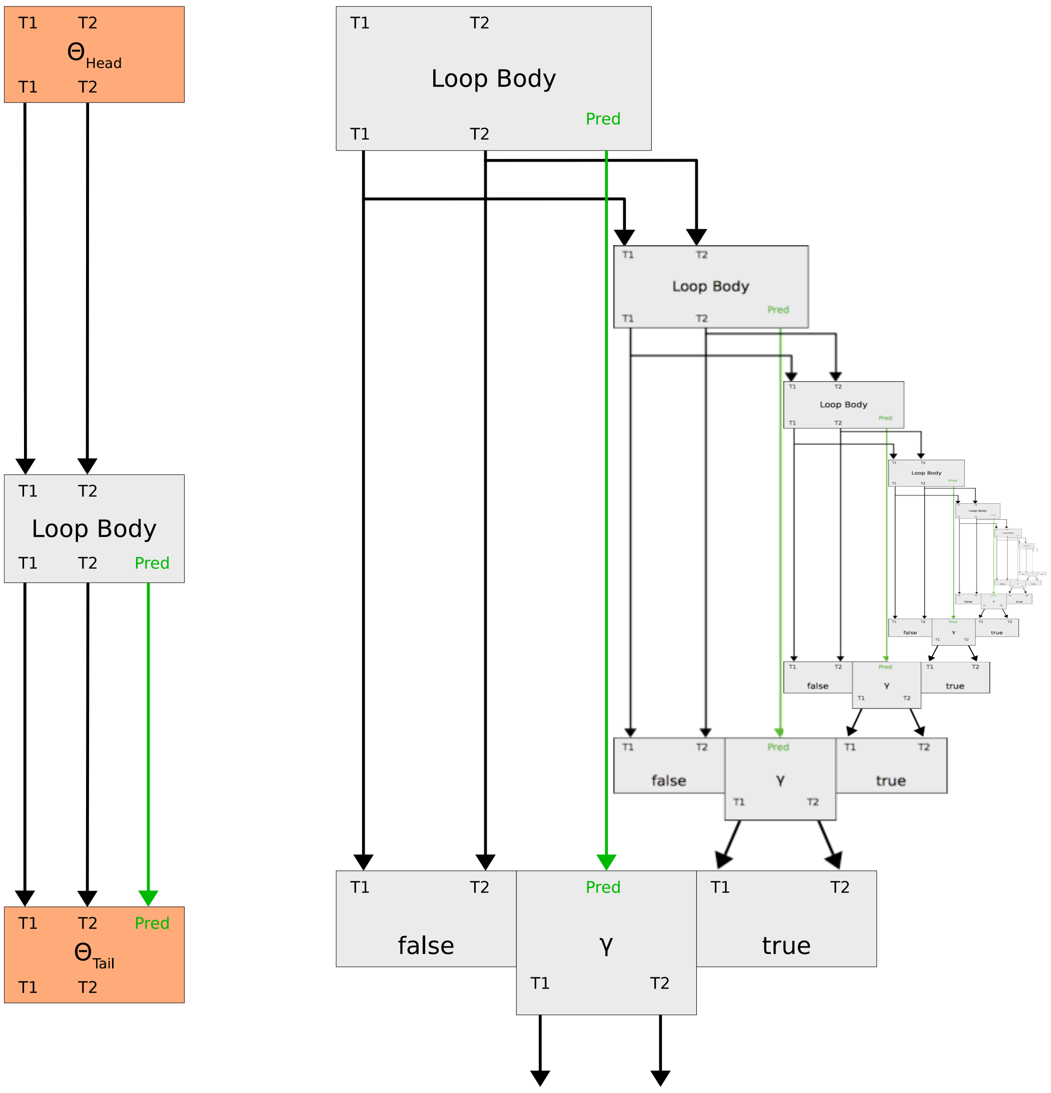

Figure 3.1: *The semantics of θ nodes.*

A special complex node in the VSDG is the θ node, shown on the left side of figure 3.1. Its purpose is to model loops. Even though the θ node is a complex node representing the distinct graph on the right side of figure 3.1, the graph will never be substituted for the node itself, since it could be *infinite* (i.e. the loop is non-terminating). The semantics of the θ node shown is that of a tail controlled loop. Other semantically alternatives for θ nodes have been proposed [33].

3.2 The Value State Dependence Graph by Example

Figure 3.2 shows the Value State Dependence Graph corresponding to the code in listing 3.1.

```
int32_t f(int32_t x, int32_t y, int32_t * z)
{
  *z = *z + x * y;
  return *z;
}
```

Listing 3.1: *An example in C illustrating value nodes/edges and state nodes/edges.*

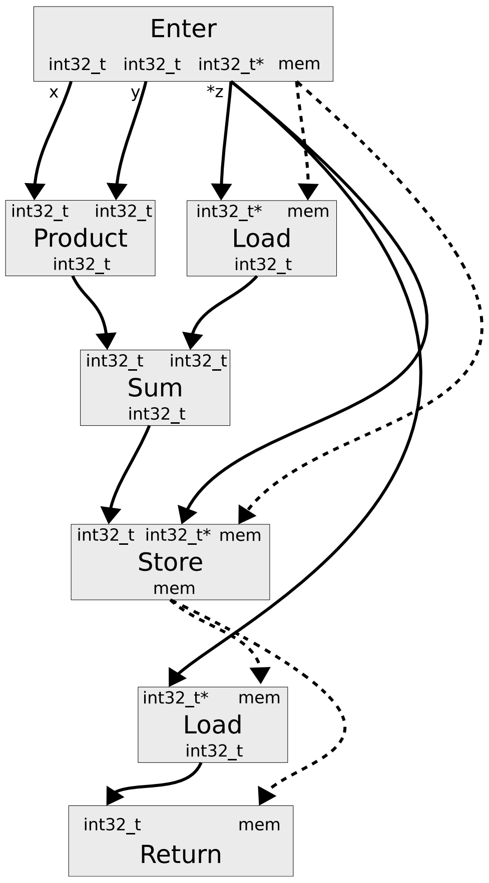

Figure 3.2: *The VSDG to the example in listing 3.1.*

The example illustrates the use of value and state nodes/edges, where value edges are drawn with a solid and state edges with a dashed line. While value nodes such as the product or sum node are automatically kept in the right evaluation order through their data dependencies, state nodes need to be kept in order with the help of state edges. Thus, it is necessary that the store node in the example consumes the old state and produces a new state which is used as the input for the second load node and therefore ensuring the intended semantics of the original program. Also notice, that the second load is actually not necessary and could be replaced by the result of the addition operation.

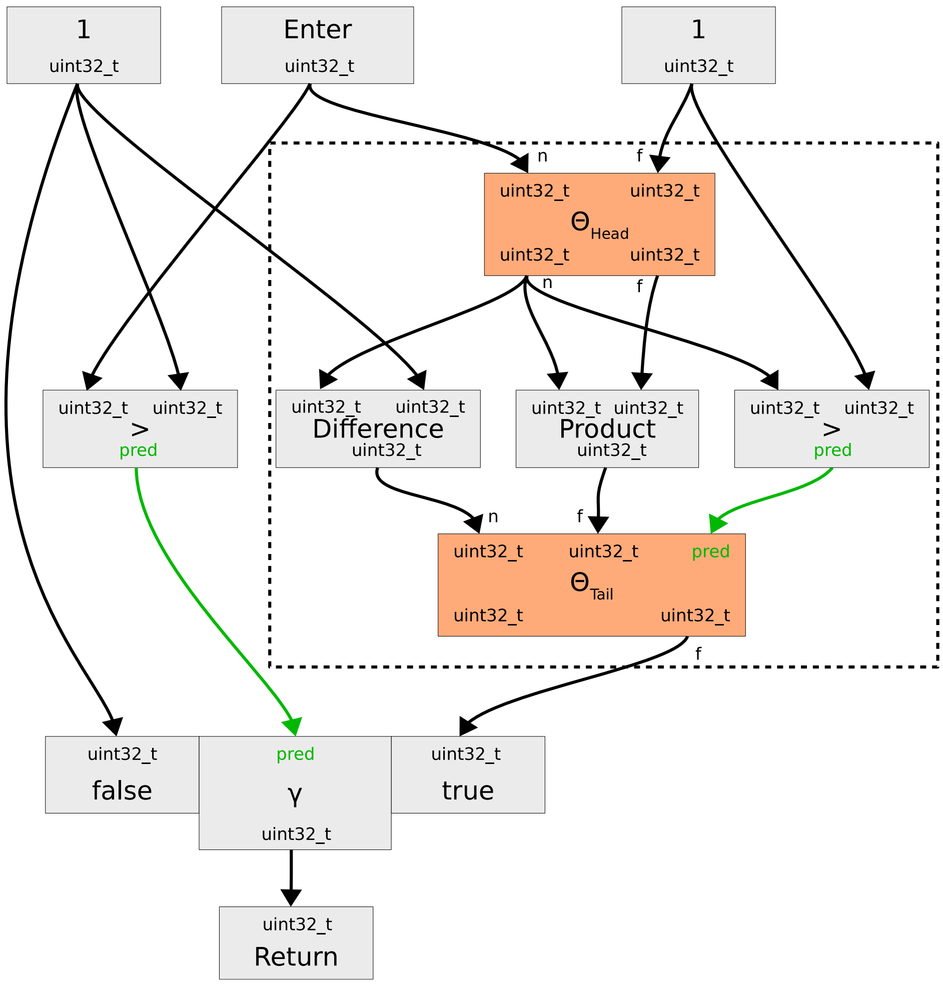

Figure 3.3: *The VSDG to the example in listing 3.2.*

The example in code listing 3.2 computes the factorial of a number and illustrates the use of γ and θ nodes. The corresponding VSDG is shown in figure 3.3. Since the for loop in C is a head controlled loop, a γ node needs to be wrapped around the θ node, with the false branch representing the case where no loop iteration is performed and the true branch containing the loop itself. The boundary of the θ node is depicted by a dashed rectangle.

```
uint32_t fac(uint32_t n)
{
        uint32_t f = 1;
        for(; n > 1; n--)
                f = f * n;

        return f;
}
```

Listing 3.2: *An example illustrating the use of γ and θ nodes.*

As you can see in this example, nodes in different branches of the graph are independent of each other, meaning that the operations represented by those nodes can be executed in parallel and thus, instruction-level parallelism is explicitly given in the Value State Dependence Graph. This makes it a perfect initial representation for code vectorization (see section 5.2.3).

3.3 Let's dance the Jive

Jive is an experimental compiler written in C which provides a complete back-end, using the Value State Dependence Graph representation as input. It is the first compiler available to the public which uses the VSDG as intermediate representation [32]. However, the VSDG is not just facilitated as it is described in section 3.1, but with a simple type system on top of it. This makes no difference semantically, but helps to catch bugs early in the translation from the front-end IR to the VSDG.

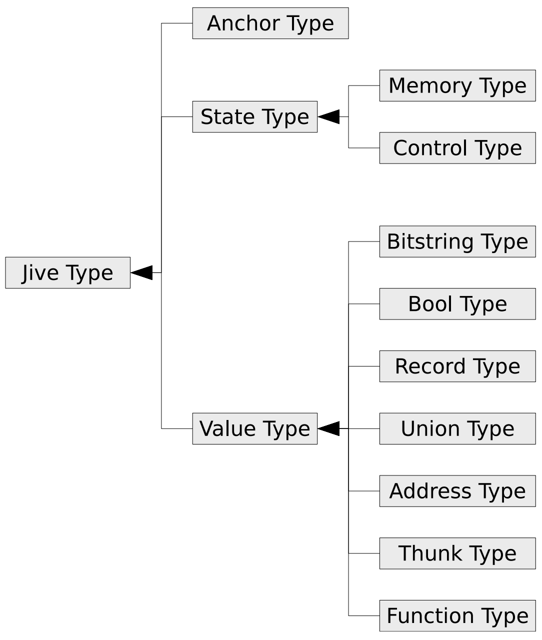

Figure 3.4: *The inheritance hierarchy of the supported types in Jive.*

A type is associated with every edge in the graph and each node checks its inputs whether they conform to the expected types, rendering the type system strongly typed. The inheritance hierarchy of all supported types are shown in figure 3.4. Three types are derived from the base type *jive type*, namely *state type*, *value type* and *anchor type*. The first two types are associated with state and value edges, respectively. As an exception to the restriction put by regions on edges, the anchor type is used for edges connecting an inner region to its parent region. The anchoring of child regions to its parent regions could have been implemented differently, but it turned out that using edges is convenient, since it is uniform with respect to the rest of the framework. The subtypes of the value and state type will be further explained along with the operations in section 3.3.1.

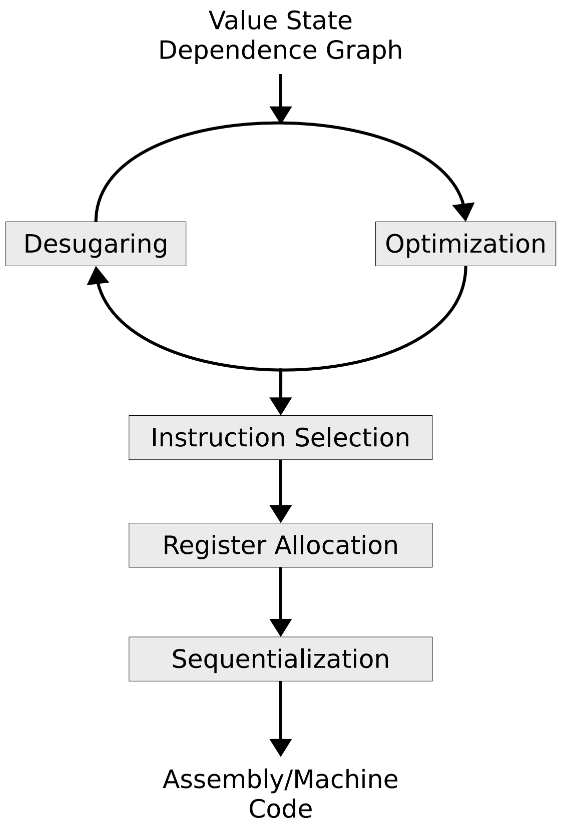

Figure 3.5: *The Jive compilation process.*

The compiler itself operates on the VSDG in roughly 5 stages, as depicted in figure 3.5. The stages are desugaring, optimization, instruction selection, register allocation and sequentialization. The desugaring stage transforms more complex nodes into sets of simpler ones. A node is said to be *primitive* if it cannot be further transformed and therefore needs to be mapped to concrete assembly/machine instructions. Additionally, Jive also offers nodes where the desugaring is language/implementation specific and therefore no routine for a transformation into simpler nodes is provided. An example would be the nodes for the support of thunks. Jive cannot know the layout of a thunk, since this is highly language and implementation specific. However, it is important to offer those kind of nodes in order to support certain kind of optimizations, for example, in the case of thunks strictness analysis [24]. Hence, as it is shown in figure 3.5, the desugaring and optimization stages are interweaved, since different kind of optimizations are only possible at different levels of abstraction. The optimization stage itself uses graph rewriting to implement code improving techniques such as the classical ones given in 3.3.3. After all nodes are transformed into primitive ones, the last three stages take place. Instruction selection is performed by replacing a subgraph of value and state nodes with one node that represents an actual machine instruction. Finally, registers are allocated for the individual instruction nodes as outlined in section 3.3.5 and the entire graph is sequentialized in order to be able to output it as assembly or machine code. Jive currently supports only the *x86* architecture. The following sections elaborate on the supported operations and the individual stages of the Jive compiler.

3.3.1 Supported Operations

This section will only explain the operations that are necessary for the back-end in chapter 4. For further information about other available operations and further details about the mentioned operations consult Jive's documentation [1].

Bitstrings

Bitstrings are used for signed and unsigned fixed-width integers and are represented in two's complement representation in Jive. However, two additional bit values besides 0 and 1 can be used in their representation, namely X and D, where X means undefined and D defined, but unknown. Those are for example used in the widening of bitstring data types, where the original value gets preceded by an appropriate number of Xs. Thus, an 8 bit value can be extended to a 16 bit value by concatenating it with 8 Xs.

The use of bitstrings for signed and unsigned integers renders the literals indistinguishable from each other and therefore, only the operations used on the individual bitstrings define their actual signedness. Jive provides the usual operations for fixed-width integers, such as logic operations (*and*, *or*, *xor*, *not*), shift operations (*shift left/right* and *arithmetic shift right*) and algebraic operations (*negate*, *sum*, *product*, *hiproduct*, *quotient*, *modulo*), where hiproduct, quotient and mod exist in two variants, one for unsigned and one for signed integers. All those operations take either one or two bitstrings as input and produce one bitstring as output.

Furthermore, the usual comparison operations such as *equal*, *not equal*, *greater*, *greater equal*, *less* and *less equal* are offered. All, but equal and not equal, exist in two variants again. Those operations take two bitstrings as input and produce an output of type bool. All operations mentioned in this section are primitive and therefore need to be matched in the instruction selection stage.

[1]http://www.jive-compiler.chaoticmind.net

Records and Unions

Records and unions are the only means of Jive for creating compound data types. Each type is supported by two operations: a construction and a selection operation. For records, the constructor is called *group* and the selector *select*, whereas for unions it is *unify* and *choose*, respectively. The record constructor takes an arbitrary number of value type derived edges as input and produces one output of appropriate type, whereas the constructor for unions takes only one input and produces an union typed output. The selectors take one edge of appropriate type as input and give back one output with the type of the selected/chosen element of the compound data type. All four operations are not primitive and need to be desugared into simpler ones.

Functions

Functions are supported by the *lambda* construct in Jive. The left image of figure 3.6 shows the function definition to listing 3.2 in Jive syntax. As you can see, an (anonymous) function consists of a region connected via an (anchor typed) edge to a lambda node. This region has an enter and leave node, where both are connected through an control typed state edge. This edge is necessary in order to ensure that the enter node is always sequentialized before its corresponding leave node. Since the leave node in figure 3.3 was dependent on the enter node through the computation itself, this edge was omitted there for reasons of simplicity. However, it is not always the case that a leave node is dependent on its enter node through the computation[2]. A lambda node has one function typed output. The function type describes a function by the number and type of its parameters/return values. In case of the fac function from listing 3.2, this would be one parameter and return value of type bitstring, both with a length of 32. However, so far it is impossible to refer to this function, since no identifier was associated with it. This is the purpose of a *definition* node, which is connected to a lambda node and associates an identifier with a function.

Two different nodes exist on the caller side: the *apply* and *partial apply* node. The first one takes as first input the type of the function and expects as the following inputs *all* parameters of this function. Its outputs are the same as the ones mentioned in the function type from the first input. The second node, namely partial apply, expects as well a function type as first input, however, the following parameters need to be *less* than the ones mentioned in the function type. Note, that this node is language specific, since it desugars to a closure and Jive cannot know the layout of a closure for a specific language. Finally, in order to refer to a defined function, a *symbolic reference* node is provided. All three nodes can be seen on the right side of figure 3.6.

[2]A simple example would be an argumentless function returning a number.

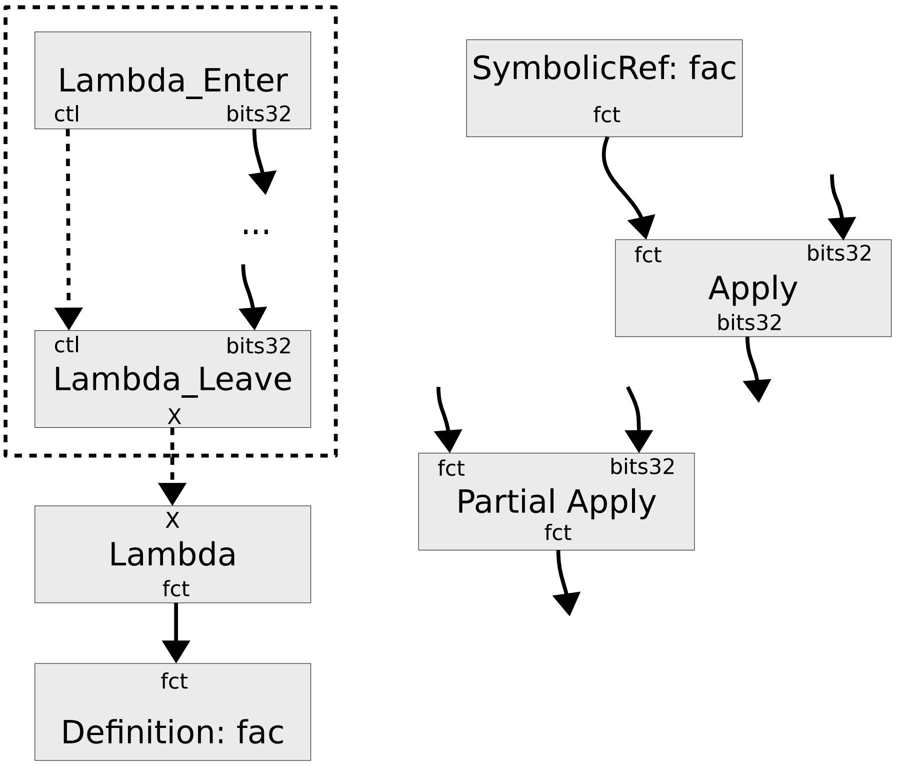

Figure 3.6: *Nodes provided for function support in Jive.*

Memory Operations

Memory operations are necessary in order to bring data from memory into registers and vice versa. Jive comes along with operations for allocating, loading and storing data as shown in figure 3.7.

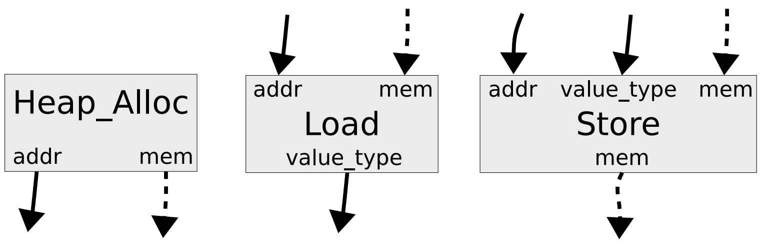

Figure 3.7: *Nodes provided for the support of memory operations.*

As its name implies, the *heap_alloc* node is used for allocating memory on the heap. It takes the size in bytes as input and has two outputs: an address (address_type) and a state (memory_type). The size could be given to it through an edge originating from a *sizeof* node. The sizeof node takes a type *as parameter* and has the size of this type as output. The heap_alloc node is language/implementation specific. It is meant for use in languages with garbage collection[3], being later replaced by a call to the allocation function shipped with the run-time system.

In order to load/store data from/to the allocated memory, load and store nodes can be used. A load node has two inputs. The first is the address (address_type) from where the data is supposed to be loaded and the second is the corresponding state (memory_type) of the address. The node has one output of a type derived from value_type.

The store node features three inputs: two value inputs and one state input. The two value inputs are the address and the data that needs to be stored. The third input is a memory_typed input and takes the state corresponding to the address given as first input. Since the store node alters external state, it also has another memory_typed state edge as output replacing the old one given as input.

Thunks

Thunks are an implementation technique for delaying the evaluation of an expression until its value is needed. They are used for implementing non-strict/lazy evaluation. Jive supports this evaluation strategy with three nodes: *thunk_create*, *force_thunk* and *thunk_value* as shown in figure 3.8.

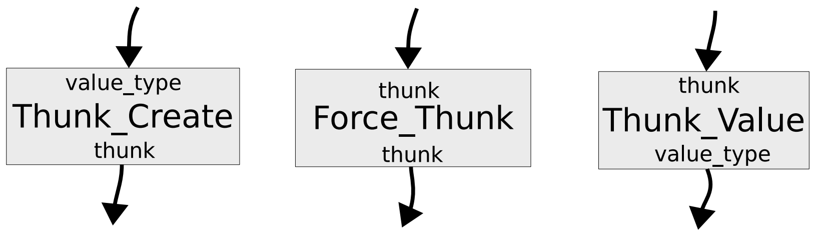

Figure 3.8: *Nodes provided for the support of non-strict/lazy evaluation.*

The thunk_create node features a value input and a thunk output. The semantics is to delay the expression (i.e. subgraph) given to it via its input by wrapping it in a thunk and returning this thunk. The opposite operation is expressed with the help of the value_thunk node. Its semantics is to *force* the thunk given as input, i.e. evaluating its contained expression, and returning the final value of the expression. Finally, the force_thunk node only forces a thunk without giving back the final value.

All three nodes are language/implementation specific and can therefore not be resolved by Jive. An example of how these nodes could be desugared is given in section 4.3.

Data Sections

Jive also provides a node for the support of global and static data. The data_object node takes one value_type derived input and depending on a parameter given, the

[3]it would never appear in a VSDG constructed from a language which uses manual memory management.

data is put at time of assembly generation into the data, rodata or bss section.

3.3.2 Desugaring

In Jive, desugaring is the act of replacing non-primitive nodes by other semantically equivalent nodes. These nodes used as replacement can itself be non-primitive again, meaning that they need to be desugared at a later point as well. The process is repeated until all nodes present in the graph are primitive.

All primitive nodes supported by Jive are bitstring typed at their value inputs and outputs and since the type system is strongly typed, explicit type conversion nodes must be temporarily inserted into the graph in order to convert types to their bitstring counterparts. The process is shown for address types in figure 3.9 from left to right. Assuming that the graph is visited top down, the upper apply node would be handled first. It would be replaced by an equivalent apply node with a bitstring output. In order to connect this node again to the second apply node, a type conversion node, namely *bitstring_to_address*, needs to be inserted. On visit of the second apply node, it is replaced by an equivalent apply node with bitstring input and as predecessor an *address_to_bitstring* node. The two consecutive type conversions are inverse to each other and can therefore be annihilated.

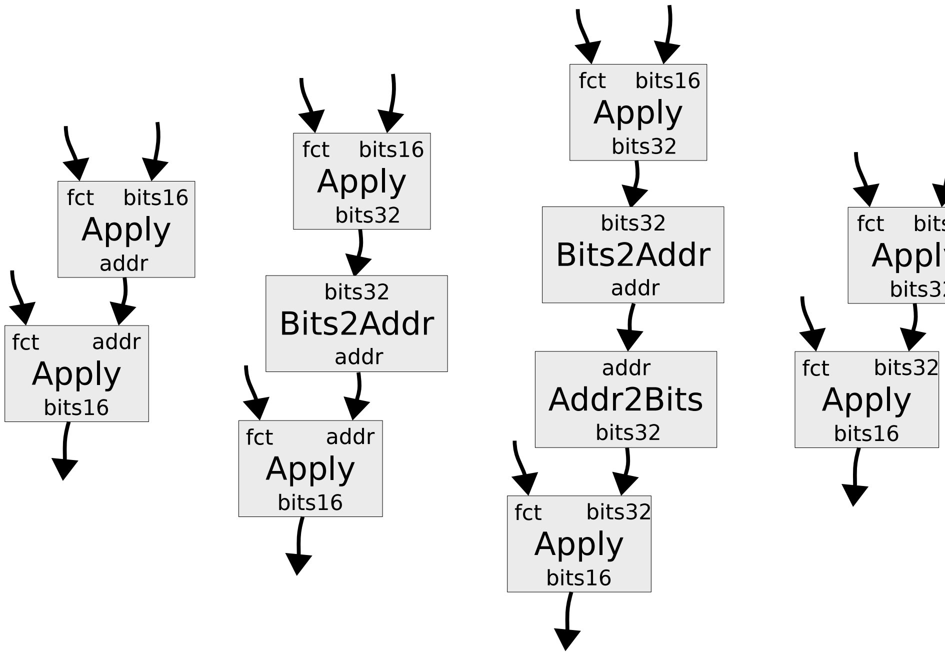

Figure 3.9: *The process of type conversion in Jive.*

3.3.3 Optimizations

Optimizations in the Value State Dependence Graph are done by traversing the graph and replacing subgraphs with an equivalent alternative which is considered better by some criteria. Several classical optimizations can be easily implemented with this graph rewriting approach. For example, dead node elimination is done by removing all nodes from the graph that have no path to a node $N \subseteq N_\infty$, meaning that they do not account for the result of any needed computation. This is the same as dead code elimination and unreachable code elimination in a classical control flow graph. Another easy to implement optimization is common subexpression elimination (CSE). Nodes representing equivalent operations with the same input operands are merged by diverting the inputs of one of two nodes consumers to take the outputs of the other node, leaving one node without any consumers. This node is then a dead node and can be eliminated by the dead node elimination pass. Note, that CSE in a Value State Dependence Graph is more generic than its counterpart in a control flow graph. It cannot just merge individual nodes but entire subgraphs which include γ and θ nodes. Further optimizations that rely on graph rewriting and are supported by Jive are strength reduction [1], arithmetical simplifications (a+0=a) and the annihilation of inverse operations. An example for the last optimization would be a group node followed by a select node. Both nodes can simply be removed from the graph and the consumer of the select node takes the original element of the group node as input.

3.3.4 Instruction Selection

Instruction selection is done by replacing primitive nodes with nodes that represent actual machine instructions of an architecture. The replacement is not necessarily done on a one-to-one correspondence, but rather on a base of subgraphs: one subgraph with only primitive nodes is replaced by a semantically equivalent subgraph containing only nodes corresponding to machine instructions. Since subgraph isomorphism is known to be NP-complete [8], a greedy strategy is employed in Jive. The graph is traversed bottom-up, on each occurrence of a primitive node, the node itself and its producers are matched against predefined patterns. On the first match, the subgraph is replaced by a semantically equivalent subgraph containing only architecture specific nodes. Generally, Jive follows the maximal munch principle by testing patterns which contain more abstract nodes first, and thus, trying to replace as many nodes in the graph as possible. For this reason, an appropriate prioritization of the individual matching subgraphs is necessary beforehand. However, this is of course highly implementation and architecture specific.

3.3.5 Register Allocation

The register allocation stage assigns a machine register to input and output ports of each instruction node. The stage consists of two passes: a preparatory "graph shaping" pass and the actual assignment of registers. The graph shaping pass partitions the graph into individual layered cuts such that a register can be assigned to each input/output port of the nodes and all value edges that pass through this cut. The basic idea of the algorithm is to attempt a depth-first traversal of the graph for the left-most values in order to have its computational dependency tree assigned to cuts as deep as possible, picking only nodes that contribute to this particular computation. The algorithm then fills up the remaining space within the cut with other computations that can be interleaved with the first one. If it is necessary, live range splitting is applied.

The second pass builds the interference graph of register candidates and a Chaitin-style graph coloring algorithm [5] is used subsequently to find a suitable register for each candidate.

In contrast to other compilers, Jive attempts by partitioning the graph into layered cuts to maintain a maximum of parallelism while exhausting the register budget without exceeding it. Also, the preparatory shaping pass allows the algorithm to be more optimistic that a global assignment can be found without additional spilling than Chaitin's classical algorithm. An alternative approach for register allocation and instruction selection for the Value State Dependence Graph has been proposed [12].

3.3.6 Sequentialization

The sequentialization pass arranges the independent nodes of the individual layered cuts, and the cuts into a sequential order. This is done by using state edges which connect the individual nodes and cuts to each other, introducing dependencies between them. However, special care must be taken for regions. All nodes in a region belong logically together and therefore must be sequentialized as a "block" with no other node from an outer region interleaving them. The final result is a unique path from a node $N_1 \subseteq N_0$ to a node $N_2 \subseteq N_\infty$, covering every node in the graph. Walking this path from N_1 to N_2 and emitting corresponding instruction encodings along the way trivially generates machine/assembly code.

4 The Jive Back-end

The main goal of this thesis was to introduce a new back-end to GHC which uses the Jive compiler for code generation. However, as already mentioned in chapter 2, the Jive back-end does not use the Cmm language as input IR but the core language, and was therefore placed directly before the core language is translated into the STG language. The GHC pipeline along with the Jive back-end is depicted in figure 4.1.

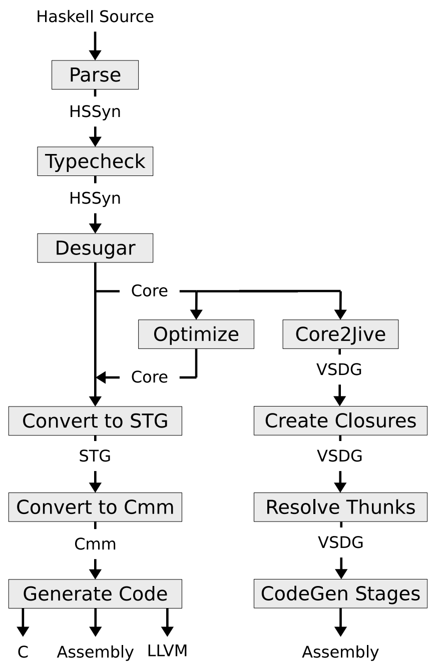

Figure 4.1: *The GHC Compilation pipeline with Jive back-end.*

The new back-end consists roughly of three stages. The first stage maps the Core language to a Value State Dependence Graph in Jive. Basically, the graph after this stage is a one-to-one mapping from Core to VSDG; it introduces nothing that was not already present in the Core representation of the program. The next stage resolves nested lambdas by creating closures for every lambda and passing this closure as additional parameter to each function, making them independent of their environment. Finally, the third stage takes care of the lazy semantics of the program by resolving the thunk nodes present in the graph. The last stage present in figure 4.1, coalesces the four stages discussed in section 3.3, namely desugaring, instruction matching, register allocation and sequentialization.

The following sections elaborate on the new stages of the compilation pipeline.

4.1 Mapping Core to Jive

This stage translates a program in the Core IR to a Value State Dependence Graph. It was implemented with the help of the Foreign Function Interface (FFI) [2] of Haskell. The FFI makes it possible to invoke code written in other programming languages, in our case C, from Haskell and vice versa. It was used to create a small interface for the necessary routines in the Jive library in order to be able to translate a program from the Core IR to a VSDG. Listing 4.1 shows a small example how the FFI can be used to interface to a C library such as libjive.

```
 1  {-# LANGUAGE ForeignFunctionInterface #-}
 2
 3  module FFIExample
 4    (
 5      JiveGraph
 6    , jiveGraphCreate
 7    ) where
 8
 9  import Foreign
10  import Foreign.C.Types
11
12  import JiveContext
13
14  newtype JiveGraph = JiveGraph (Ptr JiveGraph)
15
16  foreign import ccall "jive_graph_create"
17    jiveGraphCreate :: (Ptr JiveContext) -> IO (Ptr JiveGraph)
```

Listing 4.1: *A small example for using the FFI of Haskell.*

The important part of the listing is in line 16 and 17. A new Haskell function gets defined in these two lines whose implementation is not given in Haskell itself, but in C. We need to tell the Haskell compiler two things: the name of the function we want to interface with (jive_graph_create) and a name the function is associated with in Haskell (jiveGraphCreate) along with its type. After this declaration it is possible to use this C function as it would have been originally defined in Haskell. Further information about using the FFI can be found in [26].

Another important aspect of this translation process is the mapping of types of the original program into Jive. Most types in Haskell are *boxed*, meaning that they are represented by a pointer to a heap object. Boxed types have two type representations in Jive: a *lazy* and a *strict* one. The lazy representation is of course Jive's thunk type. These thunk types are used if a boxed type occurs as parameter or return type of a function. This ensures that the lazy semantics of the language is preserved. The strict type is only used if the inner structure of a type is needed such as after a load node. The other types in Haskell are *unboxed* types. Those types correspond to types as one would use them in C such as long int. Unboxed types have only a strict representation.

The mapping of Haskell types to their strict representation and the translation of the individual constructs present in the Core language into Jive are explained in the following sections.

4.1.1 Bindings

Bindings are used to associate a name with an expression and can be trivially translated by first translating the expression into a lambda and then give this lambda to a definition node as input, binding the lambda to the name given in the binding. An expression is always translated into a lambda node, even if it is not a lambda expression itself in the Core language. Expressions with arity zero are translated to parameterless lambdas in Jive. No special care needs to be taken for recursive bindings and they are just translated as described above one after another.

4.1.2 Literals

The only literals interesting for this thesis are those for fixed-precision signed integers. According to the Haskell 98 report [28], they need to cover at least the range $[-2^{29}, 2^{29}-1]$. Since those last 2 bits are not needed in the implementation, the range is extended to 32 bit and thus, primitive fixed-precision integers are represented by 32 bit wide bitconstants. Hence, the strict type for fixed-precision integers is a 32 bit wide bitstring.

4.1.3 Lambda Expressions

As mentioned in section 2.2, the lambda expressions in the core language are nested in order to express multi-parameter functions. This could be readily translated into Jive by creating a function closure taking only the first parameter and returning another closure that takes the second parameter and so forth. This is shown in the left image in figure 4.2. However, this would give incredibly inefficient code and the lambda expressions are therefore uncurried and translated into multi-parameter lambdas in Jive as shown in the right image of figure 4.2.

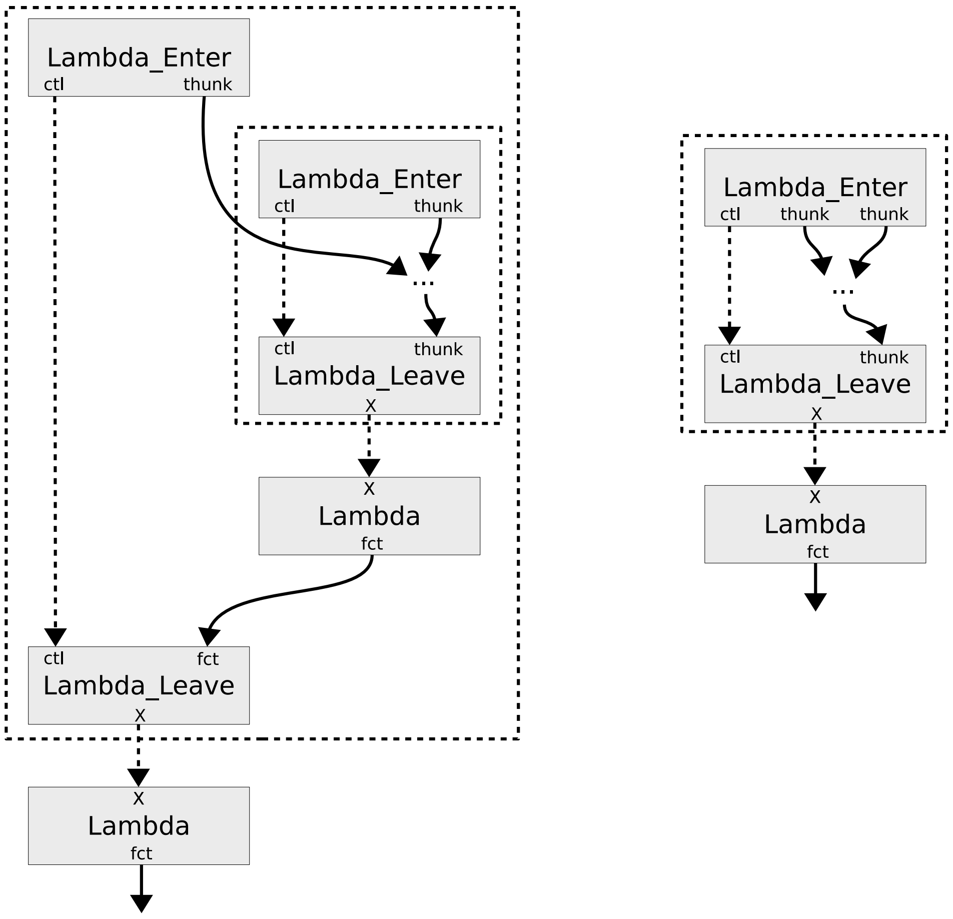

Figure 4.2: *Inefficient translation (left) vs. efficient translation of lambdas (right).*

Of course, the strict type for functions is Jive's function type.

4.1.4 Applications

Basically, an application can be performed on four different kind of Ids in GHC Core:

- Functions
- Data Constructors
- NewType Constructors
- Primitive Operations

The following four sections are each devoted to one of them and explain how an application to such an Id is translated into Jive.

Function Application

As mentioned in section 4.1.3 the lambda expressions in the core language are nested and thus, the applications must therefore be nested as well in order to type-check. For translation, the application's arguments must first be flattened and then applied to the function. The translation depends on whether the arity of the function is known at compile time or not, rendering it a *known* or *anonymous* function, respectively. If it is known, three cases are possible:

- The number of arguments provided is **less** than the arity of the function.
- The number of arguments provided **equal**s the arity of the function.
- The number of arguments provided is **greater** than the arity of the function.

In the first case, a partial apply node is created having the function and the provided arguments as input. This partial apply node is later translated into a dynamically allocated closure as explained in section 4.2. If the number of arguments equals the number of parameters, a normal apply node can be emitted, which will later correspond to a normal function call. In the last case, namely the number of arguments provided is greater than the arity of the function, an apply node is created taking the function and the number of arguments equal to function's arity as input. The result of this function call must be another function to which the rest of the arguments need to be applied. However, this is now an anonymous function and its arity is not known at compile time. Hence, we need to deal with it at run-time. The method chosen is the same as already performed in the GHC compiler: *eval/apply* [22]. Basically, the returned function plus the rest of the arguments are passed to a function (further called *rtapply*) provided by the run-time environment. Rtapply extracts the arity of a given function from its closure and applies the same procedure which was outlined above for known functions at run-time.

Since the number of arguments passed to rtapply are not fixed and can vary in number, we would need to provide for reasons of efficiency several rtapply functions which take a different number of arguments. If we still encounter more arguments than the provided rtapply functions can take, we need to chain two or more functions together in order to match the number of arguments. Another important complication is that the arguments may need to be passed to the functions in different register classes, e.g. floating-point arguments need to be passed in floating-point registers. Thus, in principle it is necessary to provide R^N functions, where N is the maximum number of arguments that could be given and R the number of register classes we need to support. Since N is in general unlimited, it is in practice necessary to identify the common cases and provide the rtapply functions for them and take care of the rest via chaining as described above. For this thesis was the simplest approach adapted: only one rtapply function per register class is provided. It takes one argument at a time and multiple arguments are resolved through function chaining. Furthermore, since only pointers and 32-bit non-pointers (i.e. fixed-precision integers) are supported in this thesis and both reside in the same registers on the x86 architecture, only one rtapply function needs to be implemented.

Data Constructors

Data Constructors arise in Haskell as part of data type definitions. A data type definition is a collection of alternatives of a fixed ordered number of types. Every alternative has a data constructor associated with it serving as a unique identifier for it. This resembles the general structure of an algebraic data type quite well: a collection of product types wrapped in a sum type with the tag of the sum type indicating the product type in use.

This general structure translates readily into Jive with a product type being nothing else than a record and a sum type being represented as a record with an integer (the tag) as first and an union as second element. The tag indicates the alternative in use in the union. The pseudo-graph in figure 4.3 shows the general creation of a value of an algebraic data type in Jive. Even though the union shown in the image takes several inputs, it only has *one* of those inputs present in a real graph. The several inputs were just drawn to symbolize the alternatives the union can take.

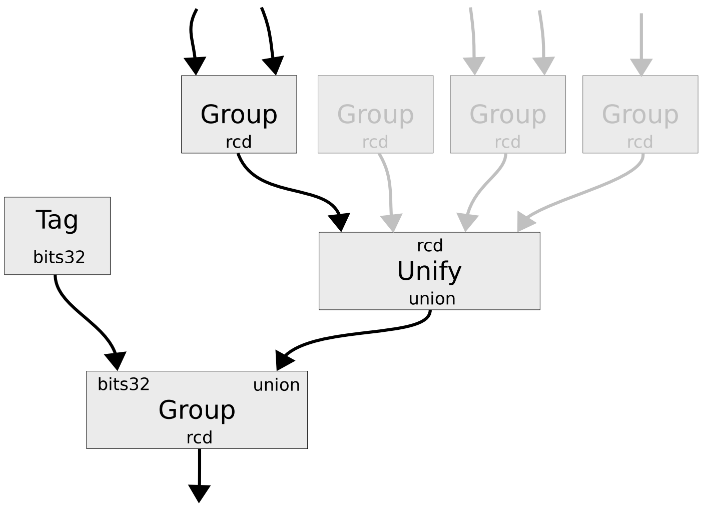

Figure 4.3: *General structure of algebraic data types.*

```
I# :: Int# -> Int
I# = Lam i (App (Var I#[1]) i)
```

Listing 4.2: *Core representation of the Int definition from listing 2.2.*

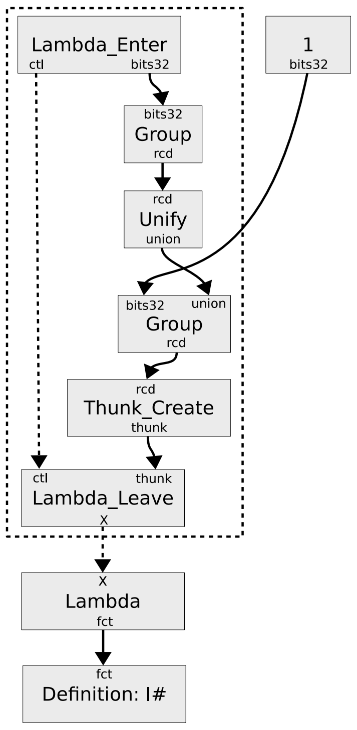

Figure 4.4: *Corresponding VSDG of listing 4.2.*

GHC creates for every alternative of a data type definition a function with the same number and types of arguments and the function's return type being that of the data type. This function serves as *constructor* of an alternative of a data type definition. An example of the desugaring of the Int data type from listing 2.2 is shown in listing 4.2, together with the translation into Jive in figure 4.4. The number in brackets indicates the tag of the data constructor in listing 4.2.

No care was taken for degenerated algebraic data types such as tuples (i.e. only one constructor) or enumerated types (many constructors with no arguments, e.g. Bool).

Newtype

A newtype constructor only exists at compile time for rendering a type distinct to the type-checker. Since a newtype can only have one constructor, it is in direct correspondence with the type it wraps up and both can be treated interchangeable at run-time. Thus, the constructor is only stripped off its argument and compilation continues as if the constructor was never present.

Primitive Operations

Table 4.1 shows the primitive operations supported by GHC for fixed-precision integers.

Operation	Type Signature	Description
(+#)	Int# -> Int# -> Int#	Addition
(-#)	Int# -> Int# -> Int#	Subtraction
(*#)	Int# -> Int# -> Int#	Low word multiply
mulIntMayOflo#	Int# -> Int# -> Int#	Return non-zero if upper word of multiply might contain useful information
quotInt#	Int# -> Int# -> Int#	Division
remInt#	Int# -> Int# -> Int#	Modulo
negateInt#	Int# -> Int#	Negation
addIntC#	Int# -> Int# -> (#Int#, Int##)	Add with carry
subIntC#	Int# -> Int# -> (#Int#, Int##)	Subtract with carry
(>#)	Int# -> Int# -> Bool	Greater
(>=#)	Int# -> Int# -> Bool	Greater or equal
(==#)	Int# -> Int# -> Bool	Equal
(/=#)	Int# -> Int# -> Bool	Not equal
(<=#)	Int# -> Int# -> Bool	Smaller or equal
(<#)	Int# -> Int# -> Bool	Smaller
chr#	Int# -> Char#	Convert to Char#
int2Word#	Int# -> Word#	Convert to Word#
int2Float#	Int# -> Float#	Convert to Float#
int2Double#	Int# -> Double#	Convert to Double#
uncheckedIShiftL#	Int# -> Int# -> Int#	Shift left
uncheckedIShiftRA#	Int# -> Int# -> Int#	Shift right arithmetic
uncheckedIShiftRL#	Int# -> Int# -> Int#	Shift right logical

Table 4.1: *Primitive operations for fixed-precision integers*

The convert operations, namely chr#, int2Word#, int2Float# and int2Double#, are not supported in this implementation, since their result type is not in the scope of this thesis. Furthermore, addIntC# and subIntC# are also not supported, since there exists no equivalent construct in Jive (yet). However, this is not a major drawback, since both operations are only needed in conjunction with full-precision integers. The missing constructs in Jive can easily be fixed by introducing addition and subtraction with carry nodes.

The normal arithmetic operations, i.e. (+#), (-#), (*#), quotInt#, remInt#, negateInt#, uncheckedIShiftL#, uncheckedIShiftRA# and uncheckedIShiftRL#, map readily to their counterparts in Jive. One arithmetic operation which maps not just to one single node in Jive is mulIntMayOflo#. Its suggested implementation on a 32-bit platform is to perform a full range multiplication and subtract the upper 32 bits of the result from the lower 32 bits shifted right by 31. This would lead to 5 nodes in Jive, namely a hiproduct, product, shift right, subtract and a bitconstant with the value 31.

Finally, the last operations in the table are the comparison operations. Those operations all have the return type Bool. Thus, in order to match their type signature, they need to be translated into a gamma node in Jive. This gamma node takes as predicate the corresponding comparison node and has as its cases the corresponding alternative of the Bool data type. However, this data type is defined in Haskell's Prelude and the order of the alternatives is not specified. This could lead to nasty bugs in programs whenever the tag used for the data constructor in Jive does not coincide with the alternative in the data type defined in the Prelude. The problem is that the developers of GHC introduced here a dependency between primitive operations and a data type defined in the Prelude. A better solution would have been to introduce a new primitive data type for booleans.

4.1.5 Case Expressions

As mentioned in section 2.2, case expressions are used for the evaluation of expressions, meaning that a case needs to force the thunk of its scrutinee and compare its value to the alternatives listed. Since this list is always exhaustive, either because all alternatives are listed or by the use of the DEFAULT alternative, an alternative is guaranteed to be taken. This scheme translates readily to the pseudo-graph shown in figure 4.5.

In this scheme the scrutinee, $scrt$, is compared to the first alternative, alt_1, and if both coincide, the expression for this alternative, namely e_1, is taken. If they are not equal, the scrutinee is compared to the second alternative, alt_2, and its expresssion taken on equality. This is continued until the second to last alternative, with the expression corresponding to the last one in the false case of this γ node. Thus, a case expression with a list of n alternatives, corresponds to a γ-*tree* with $n-1$ γs. In case of only one alternative listed, no γ node is needed and the scrutinee plus the alternative's expression is straightforwardly translated.

The nodes necessary for the comparison of the scrutinee with the alternatives depends on the type of the alternatives:

- A **DataAlt** comparison is used for comparing values that were defined with the data keyword in Haskell. The comparison is carried out by forcing the scrutinee with the thunk_value node, extracting the tag of the algebraic data type and comparing it to the one listed in the alternative.
- A **LitAlt** comparison is used for comparing literals. Since the only literals supported in this thesis are fixed-precision signed integers, the comparison is a simple bitstring equal node with the scrutinee and the literal listed in the alternative as input.

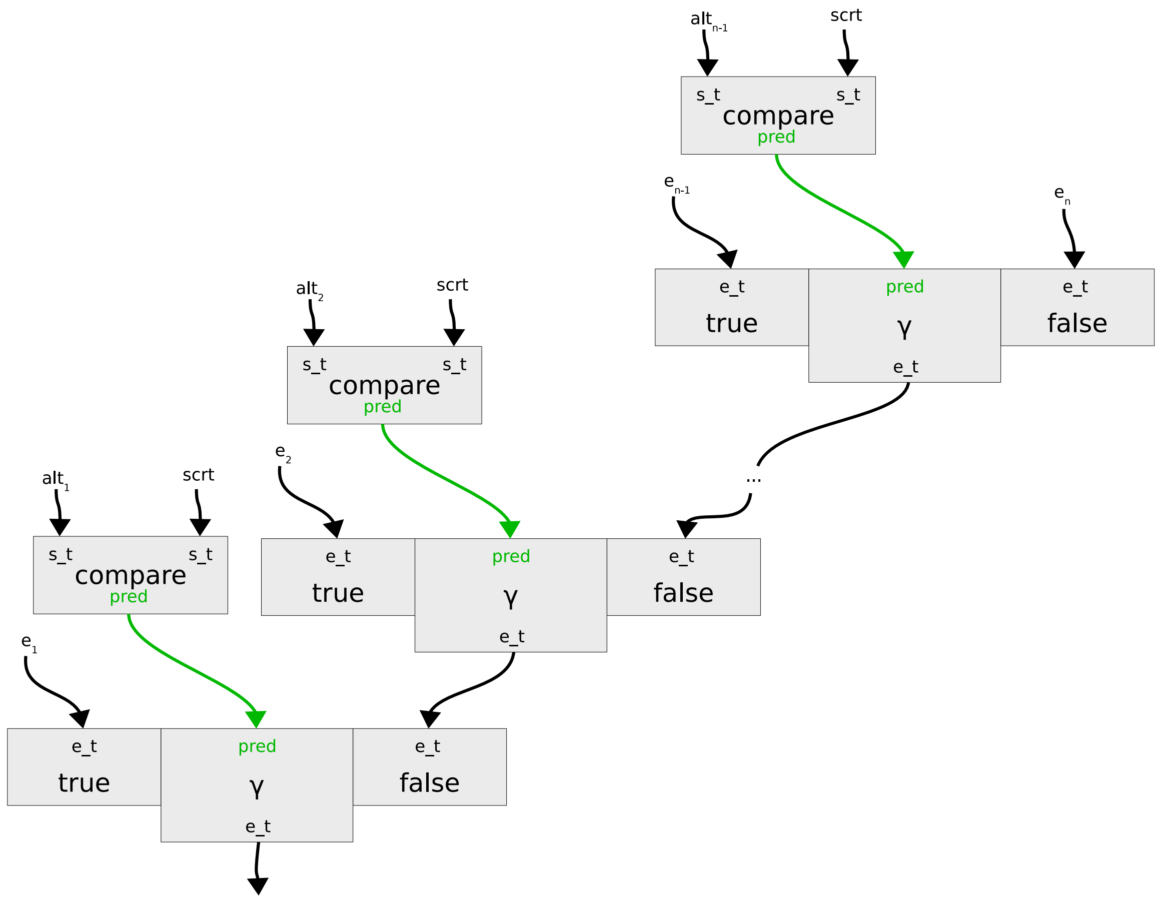

Figure 4.5: *Pseudo-graph for the translation scheme of case expressions.*

4.1.6 Cast Expressions

Casts are used for casting an expression from one type to another. An example where this is used would be pattern matching on newtypes as shown in listing 4.3.

```
newtype Int2 = Int2 Int

foo :: Int2 -> Int
foo (Int2 i) = i
```

Listing 4.3: *An example of a use case for casts.*

Since a newtype value and the value it wraps up can be treated interchangeable at run-time, the argument given to foo can be cast from Int2 to Int and returned. Thus, the behavior of foo is at run-time exactly the same as for the identify function. Cast expressions can therefore be translated by continuing the translation of their expression with the type the expression is cast to.

4.1.7 Let Expressions

Let expressions can be trivially translated into Jive, since all components for the translation are already present: First, the binding is translated as described in section 4.1.1 and then the expression itself is translated.

4.2 Creating Closures

Closures are a feature of languages that support nonlocal names, i.e. free variables, together with nested functions. A closure is a function together with its environment containing the free variables of this function. The environment allows the function to reference its free variables by extending their lifetime to the lifetime of the closure itself.

Closures can be implemented with the help of records. The first element of this record is the address to the function followed by the function's arity. The arity is needed by the rtapply function in order to be able to determine the number of arguments that need to be applied to the function at run-time. The free variables of a function follow its arity as elements in the closure. The entire layout of a closure is shown in figure 4.6.

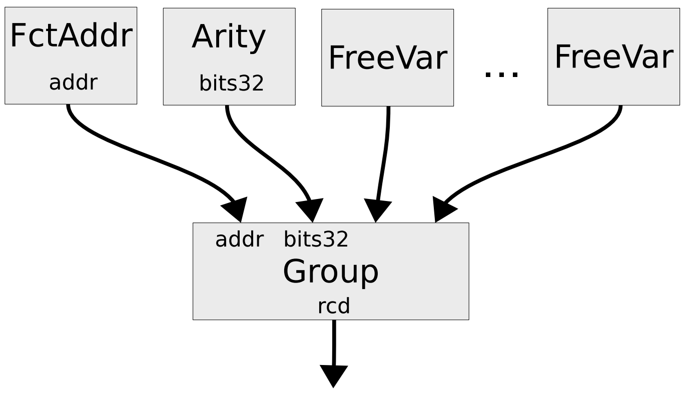

Figure 4.6: *Layout of a closure.*

In order to be able to store the free variables of a function in its closure, they first need to be identified in the Value State Dependence Graph. Basically, a variable from the original program is nothing else than a value edge in the VSDG. Thus, a free variable of a function in the Value State Dependence Graph is an value edge that originates *not* from a node that is present inside the function's region, but in a surrounding function region. This is shown in figure 4.7 with the red arrow being the free variable of the bar function.

After we identified the free variables of a function, we build its closure on the caller side and pass it as additional argument to the function. In the function itself, the values are extracted from the closure with the help of a load node and used instead of

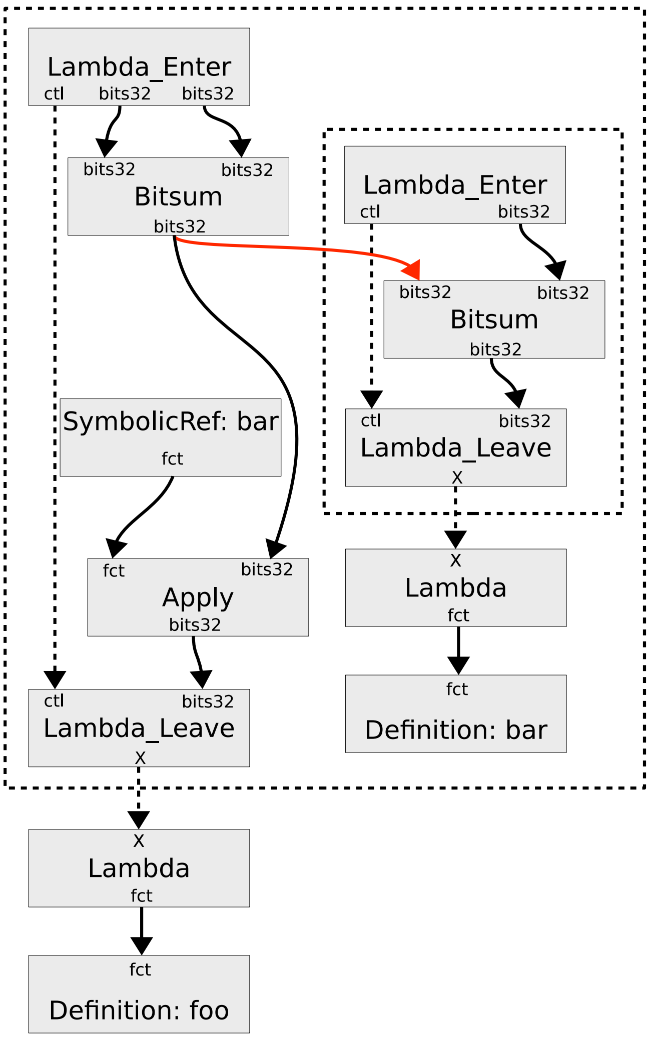

Figure 4.7: *Free variables in the Value State Dependence Graph.*

the original free variables. After this transformation, every function in the graph is independent of its environment and therefore all nested functions can be moved out of their surrounding regions to the top level. Functions that already resided on the top level have naturally no free variables. Hence, their closure only consists of the function's address and its arity. Such a closure is created in the read only data section.

In order to be able to introduce load and store nodes for closures and later thunks, state edges for sequentialization are needed. The possibly simplest scheme was adapted for this thesis: one state edge is routed through the graph representing the entire memory. This scheme has the benefit of being easy to implement and deal with, however, a heavy price is paid in terms of performance: all store nodes will be sequentialized with respect to each other. This is in most cases too conservative, since this would mean that all addresses a store is performed on would alias each other. The approach is perfectly suitable for a proof of concept as it is done here in this thesis, but would certainly not be acceptable in an industrial strength compiler.

4.2.1 Resolving Partial_Apply Nodes

Now that we know the layout of closures and how to identify free variables in the Value State Dependence Graph, it is possible to explain the resolution of partial_apply nodes. Even though this transformation is the first in the closure pass, it is explained last because the preceding knowledge about closures was necessary in order to understand this transformation.

The basic idea for transforming a partial_apply node is that it needs to be converted into a closure with the given arguments to the node being stored as free variables. This ensures that the given arguments can be applied to the function whenever the rest of the arguments are provided. Figure 4.8 illustrates the transformation process.

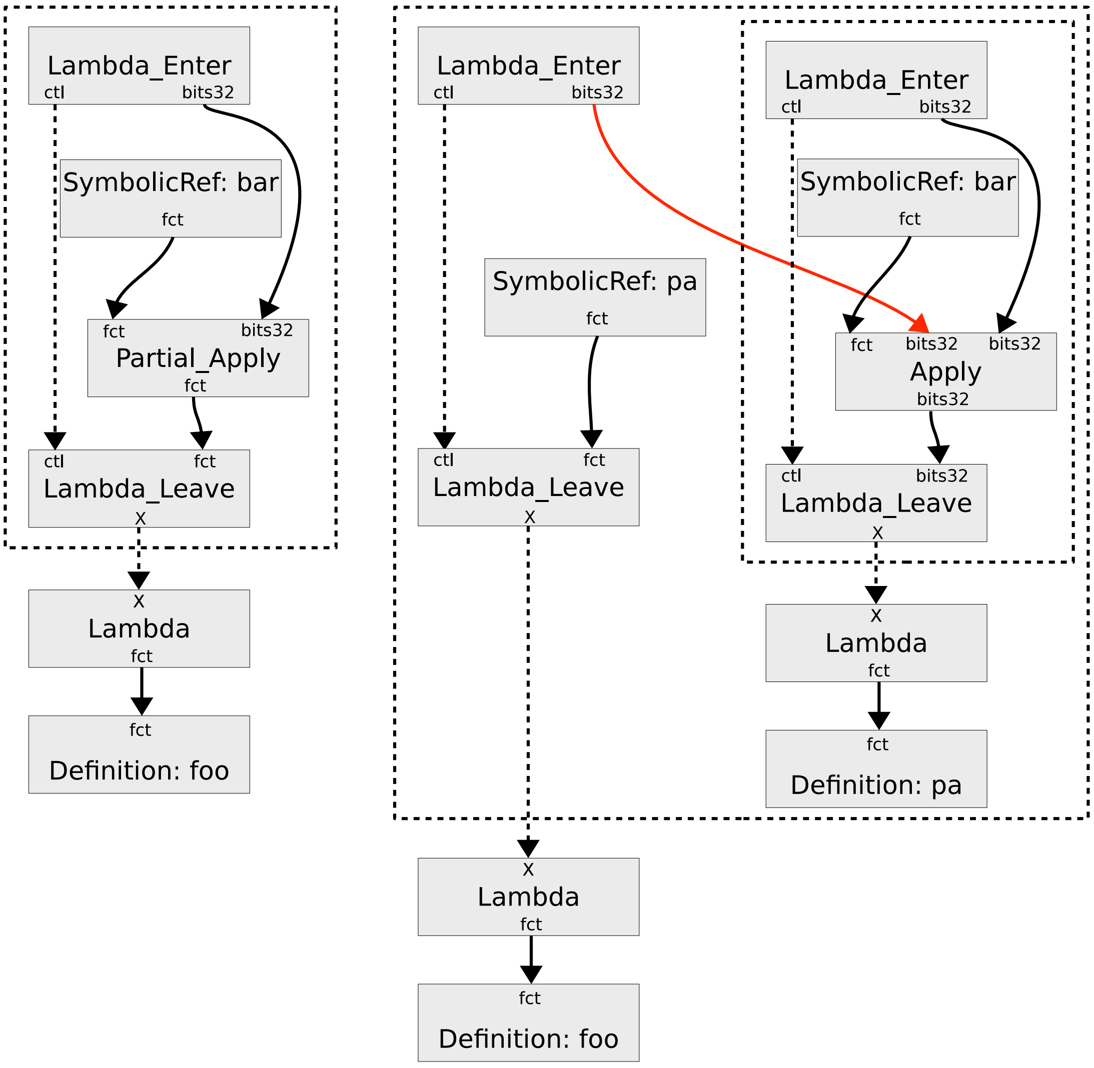

Figure 4.8: *The transformation process for partial_apply nodes.*

For each partial_apply node a function is created that has the same number and type of parameters as were arguments missing to the function of the partial_apply node.

Together with the provided arguments which are given as free variables (the red arrow in the image on the right), it is possible to call the original function with the help of an apply node. The partial_apply node itself is replaced with a symbolicref to this new created function. This symbolicref node is in a later phase replaced with the corresponding closure.

4.3 Resolving Thunks

By default, Haskell uses lazy evaluation as evaluation strategy for expressions, delaying the evaluation of an expression until its value is required and further storing the value in order to avoid a repeated evaluation. The implementation of non-strict evaluation can be achieved by wrapping the expression into a parameterless closure, called a *thunk*. However, in order to implement lazy evaluation and not just non-strict evaluation, a thunk must additionally contain a tag stating whether it was already evaluated or not and additional space must be allocated for the value of the evaluated expression. The general layout of the thunk structure used in this thesis is shown in figure 4.9. The tag can either be zero, meaning that the thunk is unevaluated and the value not present, or one, meaning the exact opposite. The tag value is always one in a thunk that has already been forced.

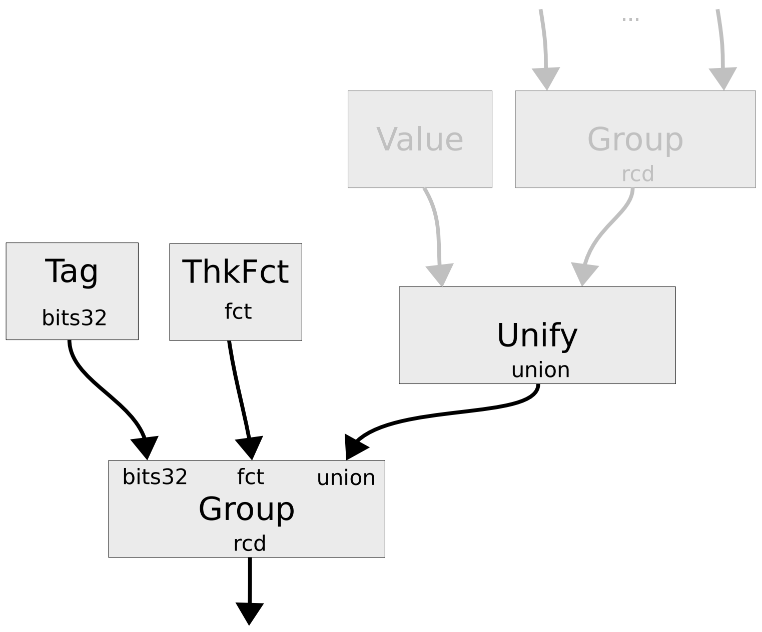

Figure 4.9: *The layout of a thunk.*

As mentioned in section 3.3.1, Jive offers three nodes for the support of thunks: thunk_create, force_thunk and thunk_value. The following sections explain how these three nodes are resolved in order to support lazy evaluation.

4.3.1 Resolving Thunk_Create Nodes

The thunk_create node's semantics is to delay the evaluation of the expression connected to it via its input. This is achieved by creating a new function for every thunk_create node, wrapping the expression into this function. The entire transformation is depicted in figure 4.10. The upper function with the containing thunk_create node is transformed into the two functions below it, with the right one being the function created for the thunk (denoted as ThkFct).

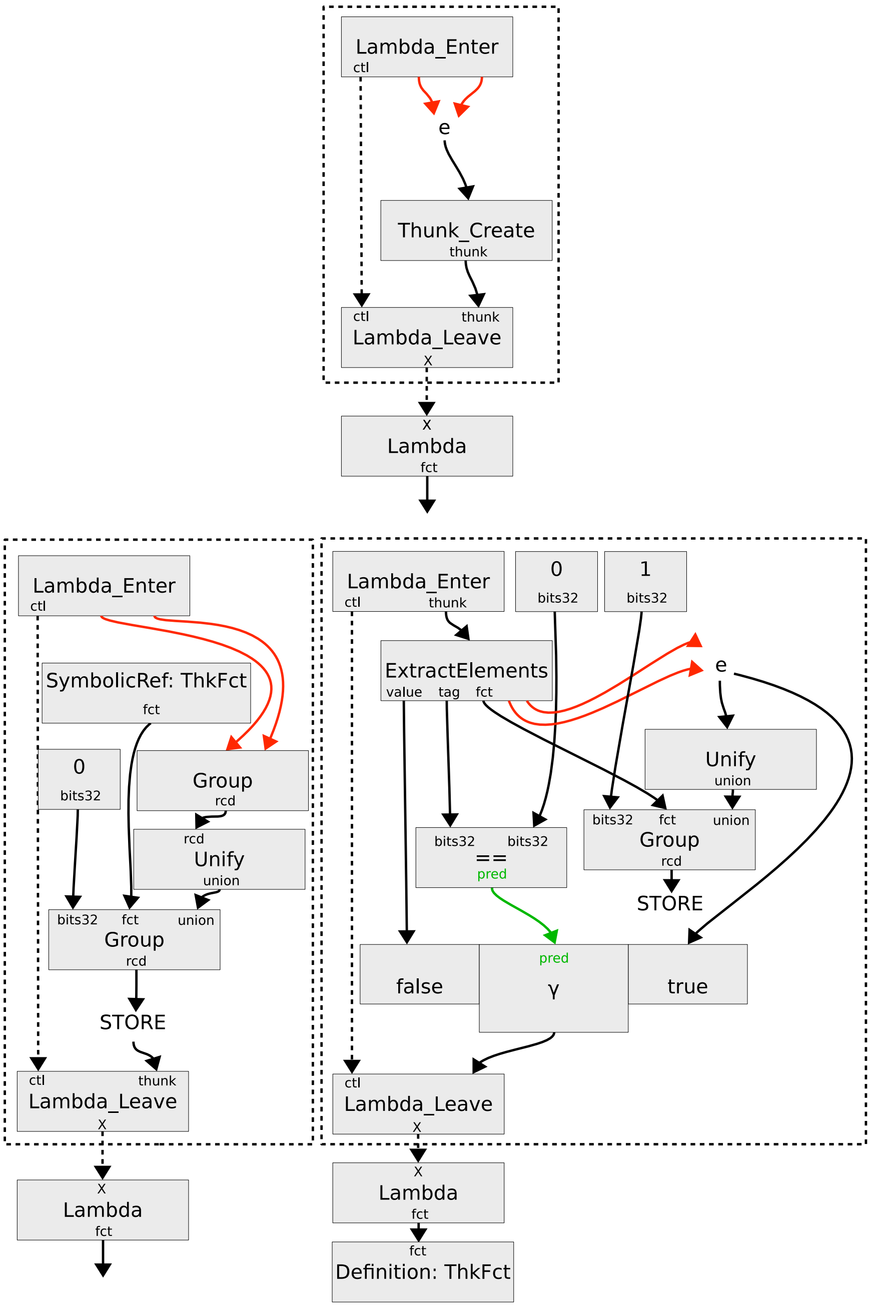

Figure 4.10: *Resolving thunk_create nodes.*

As you can see, the thunk_create node is replaced by an unevaluated (the tag is zero) thunk structure as it was shown in figure 4.9. The "STORE" below this structure means that the thunk needs to be stored in memory. This is done with the help of a heap_alloc and store node and was omitted in this image for reasons of shortage of space. The ThkFct function gets as argument the thunk structure itself. It extracts its elements, which is denoted in the figure with the "ExtractElements" node, and compares the tag. If the tag is zero, meaning that the expression is unevaluated, it evaluates the expression and creates a new thunk structure. This structure contains now a one as tag and the value of the evaluated expression. It is stored at the same address of the old structure which was given to the function as argument and therefore overriding it. Finally, the value of the evaluated expression is returned. In case the thunk was already forced earlier and therefore the tag contained in the structure is one, the value that is also present in the structure can just be returned.

Several optimizations are possible with this scheme. For example, one optimization would be to take special care of values that are already in *normal form*, i.e. fully evaluated, by not applying the heavy machinery laid out in this section to them, but by simply wrapping them in an already evaluated thunk structure. However, since this is just a proof of concept, such optimizations were not implemented as part of this thesis. Furthermore, advanced techniques such as blackholing [13] the address of the thunk function were not employed either.

4.3.2 Resolving Force_Thunk and Thunk_Value Nodes

Since the entire complexity of a thunk is encapsulated in the function created during the transformation of a thunk_create node, the desugaring of the force_thunk and thunk_value nodes are straightforward. Both only extract the thunk function from the thunk and call it. The difference is that the thunk_value node gives the return value of this call (i.e. the value of the evaluated expression) back, whereas the force_thunk node gives the address of the thunk structure itself back.

5 Summary

This chapter gives an evaluation of the new back-end along with a comparison between the already existing code generators. However, the evaluation and comparison will only be based on individual properties of the subsystems and not on experiments performed with certain benchmark suites. This is justified with the state of the current implementation which is only a proof of concept. Firstly, the implementation is incomplete in terms of supported primitive operations, in particular the operations for exceptions and IO which are needed in order to compile a "complete" Haskell program. Secondly, as mentioned on several occasions in chapter 4, it simplifies several important features that would be necessary in order to be competitive with the already established code generators in terms of performance of the produced code. Thirdly, Jive itself is only in an experimental stage and still misses several optimizations, in particular global analysis passes, in order to produce competitive code.

Some of these shortcomings are addressed in section 5.2. It gives an overview over future work regarding the back-end itself, but also about the Jive compiler. Furthermore, it also includes several suggestions about projects that are planned to be implemented in the Jive compiler of which some would also be beneficial for the GHC back-end. Finally, the last section gives a conclusion about the entire thesis project.

5.1 Evaluation and Comparison

In order to be able to evaluate and compare the new Jive back-end with the already existing code generators and even give some predictions for a future industrial strength implementation, several criteria are needed. The criteria chosen for this thesis are as follows:

- **Ease of initial implementation:** Clarifies on the required work for a full-fledged back-end and how well it would fit into the compiler.
- **Ease of future development:** Sheds some light on how much work would be necessary in order to keep the back-end up-to-date or even extend it.
- **Compilation speed:** Gives insight about the speed of compilation of the new back-end with regards to the other code generators.
- **Produced code quality:** Discusses the quality of the produced code, in particular its performance.
- **Correctness of produced code:** Deals with the amount of confidence one can put in the back-ends with regards of preserving the semantics of the original program.

The following sections are devoted to the criteria.

5.1.1 Complexity of Implementation

This section clarifies on the initial effort that is needed in order to implement a complete Jive back-end. It further touches on the cost of maintenance and thus covers the first two criteria. The proof of concept back-end implemented during this thesis project is used as a guideline throughout this section. It gives some insights for an industrial strength back-end regarding the first two criteria.

A rough estimation of the complexity of the new back-end can be obtained by having a closer look on its code size. Table 5.1 gives an overview of the lines of code for all four back-ends.

Back-End		Lines of Code
C		**1098**
NCG		**19652**
	x86	5139
	SPARC	4230
	PPC	3311
	Shared	6972
LLVM		**3469**
Jive		**4478**
	FFI	2069
	Core2Jive	1294
	C Code	1115

Table 5.1: *Lines of code for GHC's back-ends.*

As you can see, the C back-end is the smallest in terms of lines of code and therefore it is quite easy to maintain and extend it. This is due to the fact that it basically only needs to pretty print the C—— code. However, as already explained in section 2.1.3, the back-end is dependent on gcc and its *global register variables* feature used for register pinning. The necessity of register pinning neglects the benefit of platform independency by tighten it to gcc. This is especially a problem on platforms where gcc is not well supported such as Windows.

In contrast to the C back-end stands the native code generator. It comprises almost 20 times as many lines of code as the C generator, thus making it hard to maintain and extend. However, the native code generator is in contrast to the C back-end almost independent of any external infrastructure. The only tool needed is an assembler, which can be found on any platform.

The LLVM back-end is with its 3469 lines of code close to the C back-end, making it easily maintainable and extendable. It also depends as the C generator on external infrastructure, namely the LLVM compiler suite. However, in contrast to the C back-end, register pinning is better supported through a custom calling convention for GHC. Furthermore, the LLVM back-end is well supported on all major platforms and can produce code for several architectures. This makes it a good compromise between both worlds: it is relatively small and therefore easy to maintain, while it still supports all major platforms and is even able to create code for more architectures than the native code generator.

Finally, the Jive back-end comprises altogether 4478 lines of code. This makes it slightly bigger than the LLVM back-end. However, included into the count was also the bindings of the Foreign Function Interface to Jive, which account for half of the code. Since the FFI code is actually not really part of the back-end and just used as glue between Jive and GHC for the translation of the Core language into a VSDG, it could be omitted. This would put the back-end between the C and LLVM generator in terms of lines of code. This reduction in line counts can also be justified by the fact that it would have been possible to emit the code to a file as the LLVM back-end does. However, the FFI was chosen over this method for the reason of speed of compilation as it will be explained in section 5.1.2. The use of the FFI also comes with a big drawback: there is no way to check whether a defined type of a function coincides with its actual C signature. This can lead to nasty bugs as it was experienced by the author during the development of the back-end.

In contrast to the other back-ends, the Jive back-end starts off directly after the Core language. Thus, assuming that a full-fledged Jive back-end would be competitive or even superior to the other back-ends, the entire compilation pipeline from the Core stage downwards could be removed from GHC. This would be a big leap forward in terms of complexity reduction, since the entire complexity would be handled by Jive instead of GHC.

However, this is sadly just half of the story. The Jive back-end is in its present state unable to successfully compile a complete Haskell program. The reasons for that are twofold. Firstly, even though the current back-end supports all features of Haskell, it does restrict their use to programs with fixed-sized integer primitives. No other primitives are contemporary supported. Thus, a full-fledged back-end would need to support all the other primitives of GHC. The minimum requirement for compiling a trivial program with only the main function present would be the support of exception and IO primitives.

The second reason weighs in even heavier. Since the new back-end abandoned the traditional compilation path used by GHC and started off directly after the Core language, the run-time system would need to be rewritten. Contemporary, the run-time system expects the translation scheme used as defined by the STG machine. This is of course not given with a back-end that starts off directly before it. Hence, the initial implementation is the highest hurdle for a industrial strength Jive back-end, while it is expected that the cost of maintenance would be similar to the LLVM back-end.

5.1.2 Performance of the Jive Back-end

This section gives insight into the next two criteria, namely compilation speed and produced code quality. However, due to the fact that the new back-end is in its present state not competitive to the already established ones, the discussion in this section is more of a theoretical nature. Nevertheless, I am reasonable confident that experiments conducted with a competitive implementation at a later point will reflect the conclusions drawn from here.

The first criterion we are looking at is compilation speed. A good overview of it for the already established back-ends is given in the thesis [37] of the implementer of the LLVM back-end, David Anthony Terei. His experiments resulted in the NCG back-end being the fastest in terms of compilation speed, followed by the LLVM back-end being two times slower and finally, the C back-end being the slowest of them all with another two times degradation compared to the LLVM back-end. The results come at no surprise. The C back-end is clearly the slowest, since GHC needs to invoke gcc for producing assembly after it produced the C code. It is followed by the LLVM back-end, which also needs to invoke external programs in order to produce assembly code. However, the invocation of these tools seem to be way more light-weight than invoking gcc. Finally, the fastest back-end is the native code generator. This is clearly due to the fact that it does not need to invoke any external programs at all for the production of assembly code.

After we established a ranking between the individual back-ends with respect to each other, we rank the Jive back-end, or rather a future competitive implementation of it, among them. The new back-end was implemented with the Foreign Function Interface. With the help of it, the Core representation of a program is directly translated into an *in-memory* VSDG. The rest of the compilation process is then performed on this VSDG and no external program needs to be invoked before the production of assembly code. This is similar to the native code generator, which also does not dependent on any external programs. Thus, the back-end is expected to perform similar to the native code generator in terms of compilation speed. Indeed, since the code generator is written in Haskell and Jive in C, it is expected that the new-back end performs even slightly better. Furthermore, it is also possible for Jive to directly output machine instructions. This could be used in order to directly produce object code and therefore saving even the time for the invocation of the assembler.

The second important criterion in this section is the run-time of the produced code. Sadly, the implemented Jive back-end stays behind its expectations at this criterion right now. The reason for this are missing global optimizations, in particular inlining. Jive supports a lot of local optimizations such as dead code elimination, common subexpression elimination and strength reduction, but does not support any interprocedural optimizations. However, performance of the final code for the chosen translation scheme from Core to Jive relies on the fact that especially inlining is performed in Jive. Even though, GHC can perform inlining on the Core language, the let-bindings that would still be present after these transformations would be translated into functions in Jive. Without inlining, those functions cause a big overhead in the run-time of the final code. Even worse, without inlining are the local optimizations in Jive effectively not used. Thus, in order to produce competitive code compared to the other back-ends, it is necessary to implement an inliner in Jive. See section 5.2.2 for a discussion about inlining in Jive and further optimizations that can have a dramatic effect on the run-time performance of the produced code.

5.1.3 Correctness of Produced Code

This section focuses on the confidence one can put into the semantics preserving properties of the individual compilation stages and how much support is exposed by each intermediate language for catching bugs during the compilation process.

As already explained in section 2.1, a Haskell module is first parsed and type-checked and then desugared into the Core language. The Core language is a small functional language that is large enough to express the full range of Haskell. An important property of this language is that it is still fully typed. This makes it possible to check whether desugaring is implemented correctly by type-checking the Core language. Furthermore, since GHC also performs optimizations on the Core language, it is possible to check the outcome of them as well. The Core language is then translated into the STG language, another functional language. Types are mostly disregarded during this translation process. However, unlike Core, it has a well-defined operational semantics and therefore makes it possible to translate the functional program down to assembly. Thus, the STG machine serves in the GHC compiler as the link between the abstract functional world of a program and its concrete realization on hardware. With the help of the operational semantics, an STG program is then translated into C−−. At this point it is quite hard to check whether a program still corresponds to its original meaning and not much help is given, besides the weak type system of C−−, in order to check a program's validity. The rest of the translation process is then similar to compiling a C program.

Since the Jive back-end starts directly with the Core language as input and never discards types during the compilation process, it is possible to check the correctness of a program during the entire compilation process. This gives great confidence in the individual transformations performed on the VSDG. Thus, the VSDG implementation in Jive behaves in terms of the preservation of types similar to the Core language, with the big difference that it is possible to use the VSDG to produce assembly code. Furthermore, the VSDG in Jive also offers one consistent framework for the translation of Haskell. It is not necessary to define several intermediate representations with different semantics in order to convert Haskell to assembly code.

The VSDG makes it even possible to go beyond type preserving compilation, by enabling to *verify* whether a transformation is semantics preserving. What we need for this is an abstract evaluation semantics for the VSDG:

The evaluation of *one* function in the VSDG is done by replacing all its parameters through constant nodes. The evaluation is carried out until only the return node with connected constants are left. It starts with the return node and checks at each step the type of the node. If it is a/an

- **Operation Node:** Check whether all producers of the node's inputs are constants. If this is true, evaluate the node with its represented operation and replace all outputs through the constant representing the result. If it is false, choose a producer that is not a constant as new evaluation target and proceed.
- γ **Node:** Check the predicate whether it is a constant. If this is true, replace the node's outputs with the true/false case, respectively. If the predicate is not a constant, choose it as new evaluation target and proceed.
- θ **Node:** Perform one loop unrolling step, meaning that the outermost iteration is transformed to a γ node, while the following iterations are represented by a θ node that was moved inwards. See image 3.1 for the semantics of the θ node.
- **Call Node:** Replace the node through its corresponding function definition.

A transformation from a graph G to a graph G' is semantics preserving whenever for all possibilities of parameters the results of the evaluation of G' is equivalent to the evaluation of G. Since every transformation in the Value State Dependence Graph is based upon graph rewriting, it is not just possible to formally verify the stages necessary for producing assembly code, but also all optimizations.

5.2 Future Work

This section is devoted to planned future work regarding the extension of the prototype back-end to a full-featured or even industrial strength back-end. It also gives further insight into some other ideas for what the Jive compiler could be used.

5.2.1 Full-featured GHC Back-end

In order to equip GHC with a full-featured back-end, several issues must not just be addressed in the prototype back-end itself, but also in Jive. The only supported data type at the moment in Jive are fixed-precision integers. Jive must at least be extended to support single and double precision floating points in order to support a full-featured back-end. The rest of the primitive operations necessary is expected to either be covered by a combination of the already present features in Jive or by a function supplied by the run-time system.

The biggest change necessary to the prototype back-end would be the addition of further primitives. This means mainly to come up with a mapping from a primitive in GHC to a subgraph in Jive that follows the semantics of the primitive while retaining its type signature. Most of those mappings are straightforward as it was shown with the primitive operations for fixed-precision integers.

However, there are other issues that were absolutely ignored during this thesis. For example, it is necessary to provide further information along with the data structures used for closures and thunks in order to aid garbage collection. It is also expected that further issues arise when the Haskell code is compiled with support for multithreading. However, none of those topics were further explored (yet) and it is therefore hard to make accurate predictions for necessary future changes.

5.2.2 Optimizations

Jive supports currently only local optimizations that work within the scope of one function. Most of them are more general than their counterparts in a CFG and encompass several classical optimizations. For example, common subexpression elimination (CSE) does not only work on simple operations, but also on complex ones such as γ and θ nodes, and therefore exceeds the basic block boundaries in a traditional CFG. It is even possible to use CSE in the Value State Dependence Graph to merge equivalent functions, since they are also just expressed via complex nodes. Even though these optimizations are more flexible and general than their CFG counterparts, they lie at waste with the current prototype back-end. The reason for this is the lack of global analysis passes. Several such passes would help tremendously to increase the quality of the produced code:

- inlining
- strictness analysis
- tail-recursion elimination
- escape analysis

The following sections give a brief overview of these passes and their realization within Jive.

Inlining

In principle, inlining is dead simple: a function call is replaced by an instance of the function's body. However, in practice one issue needs to be taken into account: code bloating. This issue can be conveniently addressed in Jive. Since the number of nodes in a function's body reflects the number of instructions later produced, it can be used to give a rough estimate of the code bloating expected after inlining. This number can help to guide the inlining and restrain exorbitant code duplication.

Strictness Analysis

Strictness analysis is used to prove whether a function is strict in one or more of its arguments. A strict argument is guaranteed to always be evaluated by the function and hence the evaluation can also be done on the caller side instead. This can lead to a significant decrease in thunks in the program and therefore improved performance, since unnecessary evaluation of them can be eliminated. Strictness analysis in Jive is obviously based on the three thunk nodes presented in section 3.3.1. A function is strict in its argument if it always uses a force_thunk or value_thunk node on this argument. Once an argument is known to be strict, the force_thunk or value_thunk node can then be moved to the caller sides of the function.

Tail-Recursion Elimination

A function is said to be tail-recursive if the last operation performed in its body is a call to itself. Such a function behaves semantically like a loop and can therefore be replaced by one. In Jive this would mean that the body of this function would be completely replaced by a semantically equivalent θ node. This replacement could lead to further optimizations with regards to loops and would also offer a higher chance for vectorization as explained in section 5.2.3.

Escape Analysis

Escape analysis is a method for determining the dynamic scope of pointers. It can be used to check whether the address of a heap allocated object ever leaves the scope of the function where the allocation is performed. If it does not escape its scope, the heap allocation could be replaced by an allocation on the stack. In the case of Jive, this would mean that a heap_alloc node would be replaced by a stack_alloc node.

5.2.3 Vectorization

Another desirable extension for Jive would be automatic vectorization. The Value State Dependence Graph is especially suitable for this kind of program transformation, since it explicitly exposes the parallelism present in a program. The basic observation is that operations residing on different paths in the VSDG are independent of each other and can therefore be exectuted in parallel. Hence, the overall goal of a vectorizer would be to produce cuts in the graph that contain as many scalar operations of the same kind as possible without exceeding the number of slots of the vector operations. After the individual scalar operations have been arranged into cuts, it is trivial to convert them to the corresponding vector operations. Vectorization can be used to boost the performance of critical program parts such as loop bodies. However, in order to obtain enough scalar operations for filling the slots of an vector operation, several other optimizations, e.g. loop unrolling, are additionally necessary. Vectorization for the Value State Dependence Graph was already implemented in Jive's predecessor, the dynpft compiler, and successfully tested with the Cell Broadband Engine [31].

5.2.4 Source-to-Source Compilation

Another interesting application for Jive would be source-to-source compilation. As we have seen in this thesis, Jive allows for representing a program at several levels of abstraction and gives the possibility to lower this level until assembly or even machine code can be produced. However, it is perfectly fine to intercept this compilation process at a certain level and transform the VSDG into another high-level language. A first impression of such a translation was given in appendix B of Alan C. Lawrence's thesis [20]. He showed that it is possible to encode the Value State Dependence Graph in Haskell with the help of the state monad. The encoding is based on the observation that the state monad enforces the same dynamic property of state, namely every state produced is consumed exactly once, as it is necessary for well-formed VSDGs. The stateful operations are then combined with the help of the bind operator and thus sequentialized as the state edges do with operations in the VSDG. It would certainly be an interesting undertaking to add such a feature to Jive and open the compiler up for source-to-source compilation. Of course, Haskell is not the only option imaginable as target language. However, due to its close relation to the VSDG through its functional nature and the support of (state) monads, it is certainly one of the more interesting once.

5.3 Conclusion

This thesis describes the proof of concept implementation of a new back-end for the GHC compiler. The back-end is based on Jive, which is a new compiler that uses the Value State Dependence Graph as intermediate representation of choice. Originally it was planned to place the back-end at the same position in the compilation pipeline as all the other code generators. However, after a thorough investigation of the pipeline and the already present back-ends as given in chapter 2, it was clear that this would not give any benefits over the other code generators. The reasons for this are mainly twofold:

- The new back-end would inherit the idiosyncrasies and restrictions of the STG machine as all the other code generators do.

- It would not have been a thesis about compiling Haskell with the help of the Value State Dependence Graph but rather C, or more precisely C−−. However, since C is not free of side-effects, the operations in the VSDG would have been sequentialized through state edges. Thus, an alias analysis pass would have been necessary in order to (partially) recover the purity that was already present in the Haskell source in the first place. Even if this pass would have done a good job, some artificial sequentializations would have remained and restricted the Value State Dependence Graph in deploying its full power.

Thus, the new back-end was placed right before the STG machine in the compilation pipeline, using the Core representation as input language. However, in order to successfully convert Core to a Value State Dependence Graph, Jive needed to be lifted to the same level of abstraction as the Core language. Hence, several features were added to Jive with the final results of this process outlined in chapter 3. Specifically were

- functions, i.e. lambda, apply, partial apply, symbolic reference and definition nodes,
- records and unions,
- the heap alloc node
- and thunk support

added. After those features were at hand, it was possible to translate Core to a Value State Dependence Graph and convert this graph incrementally to assembly code. The translation itself can roughly be broken down into 3 stages:

- Convert the Core representation into a Value State Dependence Graph. This translation is basically a one-to-one mapping from Core to a VSDG.
- Creating Closures. This stage creates for every function a closure and resolves the partial apply nodes.
- Resolving Thunks. This stage resolves all three thunk nodes by creating a function for every thunk_create node and calls to this function for the other two nodes.

The details of those transformations are explained in chapter 4. After all language specific nodes have been transformed, the transformation passes in Jive can take over and translate the graph to assembly code. With the proof of concept back-end finally in place, it was possible to give an evaluation of it and additionally predictions for a future industrial strength back-end. Even though the comments given were only of theoretical nature, it is expected that experimental results at a later point will reflect the conclusions drawn.

In regards of implementation complexity is the new back-end initially a big hurdle, since the run-time system needs to be rewritten in order to match with the newly chosen translation scheme. However, it is expected that this will pay off in terms of the other explored criteria, in particular code quality. The new back-end shakes off the idiosyncrasies and restrictions from the STG machine, while it makes it possible to harvest to full power of the Value State Dependence Graph through the use of Core as input. I am certain that after most of the global analysis passes, which were outlined in section 5.2.2, are implemented, a code generator using Jive will not just be competitive to the other back-ends, but even superior. Furthermore, the use of Jive will also raise the confidence that can be put into the compiler with regards of preserving the semantics of the original program.

In closing, I believe that a back-end using the Value State Dependence Graph as offered by Jive is a compelling alternative. It provides a new way of compiling Haskell to assembly code. Albeit this way may be stony in its beginnings, I strongly believe that this additional effort will with certainty pay off at a later point. The Value State Dependence Graph makes it not just possible to introduce new optimizations to Haskell, but also raises the confidence that can be put in the entire compilation process in terms of semantics preservation. Furthermore, it is expected that the cost of maintenance of the back-end will be similar to the LLVM back-end, since most changes would happen in Jive itself and thus, the back-end would automatically benefit from them. Going forward I believe that I showed with this thesis the potential of the new back-end. It is a viable and worthwhile option to continue on this work by equipping GHC with an industrial strength version.

Bibliography

[1] A. V. Aho et al. *Compilers principles, techniques, and tools*. 2nd ed. Addison-Wesley, 2007.

[2] M. M. T. Chakravarty et al. *A Primitive Foreign Function Interface*. 2004. URL: `http://www.cse.edu.au/~chak/haskell/ffi/`.

[3] Frances E. Allen. “Control flow analysis”. In: *SIGPLAN Not.* 5 (1970), pp. 1–19. ISSN: 0362-1340.

[4] Clem Baker-finch, Kevin Glynn, and Simon Peyton Jones. “Constructed Product Result Analysis for Haskell”. In: *Journal of Functional Programming*. 2000, p. 2004.

[5] Gregory Chaitin. “Register allocation and spilling via graph coloring”. In: *SIGPLAN Not.* 39.4 (Apr. 2004), pp. 66–74. ISSN: 0362-1340. DOI: `10.1145/989393.989403`. URL: `http://doi.acm.org/10.1145/989393.989403`.

[6] James Cheney and Ralf Hinze. *First-class phantom types*. Tech. rep. Cornell University, 2003.

[7] Philippe Codognet and Daniel Diaz. “wamcc: Compiling Prolog to C”. In: *12th International Conference on Logic Programming*. MIT PRess, 1995, pp. 317–331.

[8] Stephen A. Cook. “The complexity of theorem-proving procedures”. In: *Proceedings of the third annual ACM symposium on Theory of computing*. STOC ’71. Shaker Heights, Ohio, United States: ACM, 1971, pp. 151–158. DOI: `10.1145/800157.805047`. URL: `http://doi.acm.org/10.1145/800157.805047`.

[9] Ron Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *ACM Trans. Program. Lang. Syst.* 13 (1991), pp. 451–490. ISSN: 0164-0925.

[10] David Gudeman, Koenraad De Bosschere, and Saumya K. Debray. “jc: An Efficient and Portable Sequential Implementation of Janus”. In: *Proc. Joint International Conference and Symposium on Logic Programming, WASHINGTON DC*. MIT Press, 1992, pp. 399–413.

[11] Fergus Henderson, Thomas Conway, and Zoltan Somogyi. *Compiling logic programs to C using GNU C as a portable assembler*. 1995.

[12] Neil Johnson and Alan Mycroft. “Combined Code Motion and Register Allocation Using the Value State Dependence Graph”. In: *Proc. 12th International Conference on Compiler Construction (CC’03) (April 2003*. 2003, pp. 1–16.

[13] Richard E. Jones. “Tail Recursion without Space Leaks”. In: *J. Funct. Program.* 2.1 (1992), pp. 73–79.

[14] Simon L Peyton Jones. “Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine”. In: *Journal of Functional Programming* 2 (1992), pp. 127–202.

[15] Simon L. Peyton Jones et al. *The Glasgow Haskell compiler: a technical overview*. 1992.

[16] Simon Peyton Jones and Will Partain. “Measuring the Effectiveness of a Simple Strictness Analyser”. In: *Functional Programming*. Springer-Verlag, 1993, pp. 201–220.

[17] Simon Peyton Jones, Will Partain, and André Santos. “Let-Floating: Moving Bindings to Give Faster Programs”. In: *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 1996, pp. 1–12.

[18] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. “C–: A Portable Assembly Language That Supports Garbage Collection”. In: *International Conference on Principles and Practice of Declarative Programming*. Springer Verlag, 1999, pp. 1–28.

[19] Chris Lattner and Vikram S. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.” In: *CGO’04*. 2004, pp. 75–88.

[20] Alan C. Lawrence. “Optimizing compilation with the Value State Dependence Graph”. PhD thesis. University of Cambridge, Computer Laboratory, 2007.

[21] André L. de M. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. 1995.

[22] Simon Marlow and Simon Peyton Jones. “Making a fast curry: push/enter vs. eval/apply for higher-order languages”. In: *Journal of Functional Programming* 16.4–5 (2006), pp. 415–449. URL: `http://community.haskell.org/~simonmar/papers/evalapplyjfp06.pdf`.

[23] Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17 (1978), pp. 348–375.

[24] Alan Mycroft. “The Theory and Practice of Transforming Call-by-need into Call-by-value.” In: *Symposium on Programming’80*. 1980, pp. 269–281.

[25] J. Dias N. Ramsey and S. Peyton Jones. “Hoopl: Dataflow optimization made simple”. In: *ACM SIGPLAN Haskell Symposium 2010*. ACM Press, 2010.

[26] Bryan O’Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. 1st. O’Reilly Media, Inc., 2008. ISBN: 0596514980, 9780596514983.

[27] Simon Peyton Jones and Simon Marlow. “Secrets of the Glasgow Haskell Compiler inliner”. In: *J. Funct. Program.* 12 (2002), pp. 393–434. ISSN: 0956-7968.

[28] Simon Peyton Jones et al. “The Haskell 98 Language and Libraries: The Revised Report”. In: *Journal of Functional Programming* 13.1 (2003), pp. 1–255.

[29] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987. ISBN: 013453333X.

[30] Benjamin C. Pierce. *Types and programming languages*. Cambridge, MA, USA: MIT Press, 2002, pp. 339–361. ISBN: 0-262-16209-1.

[31] Nico Reißmann. *Code generation for pixel format transformations on a Cell processor.* Bachelor thesis. Department of Computer Science, Technische Universität Bergakademie Freiberg, 2009.

[32] James Stanier. "Removing and Restoring Control Flow with the Value State Dependence Graph". PhD thesis. University of Sussex, 2011.

[33] James Stanier and Alan Lawrence. "The Value State Dependence Graph Revisited". In: *Proceedings of the Workshop on Intermediate Representations*. Ed. by Florent Bouchez, Sebastian Hack, and Eelco Visser. 2011, pp. 53–60.

[34] Martin Sulzmann et al. "System F with type equality coercions". In: *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*. ACM, 2007, pp. 53–66. ISBN: 159593393X.

[35] David Tarditi, Peter Lee, and Anurag Acharya. *No Assembly Required: Compiling Standard ML to C*. Tech. rep. ACM Letters on Programming Languages and Systems, 1990.

[36] David A. Terei and Manuel M.T. Chakravarty. "An llVM backend for GHC". In: *Proceedings of the third ACM Haskell symposium on Haskell*. Haskell '10. Baltimore, Maryland, USA: ACM, 2010, pp. 109–120. ISBN: 978-1-4503-0252-4.

[37] David Anthony Terei. *Low Level Virtual Machine for Glasgow Haskell Compiler*. Honours thesis. Department of Computer Science and Engineering, University of New South Wales, 2009.

[38] P. Wadler. "Deforestation: Transforming Programs to Eliminate Trees". In: *ESOP '88. European Symposium on Programming, Nancy, France, 1988 (Lecture Notes in Computer Science, vol. 300)*. Berlin: Springer-Verlag, 1988, pp. 344–358.

[39] Hongwei Xi, Chiyan Chen, and Gang Chen. "Guarded Recursive Datatype Constructors". In: *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*. New Orleans, 2003, pp. 224–235.

List of Figures

List of Tables

List of Code Listings