THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Relations in Dependent Type Theory

CARLOS GONZALÍA

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
AND GÖTEBORG UNIVERSITY
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, 2006

Relations in Dependent Type Theory
CARLOS GONZALÍA
ISBN 91-628-6763-6

Technical report no. 14 D
Department of Computer Science and Engineering
Research group: Programming Logic

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

Printed at Chalmers, Göteborg, Sweden, 2006

To the memory of my parents

De toda la memoria, sólo vale
el don preclaro de evocar los sueños.

Antonio Machado, *Galerías*

Abstract

This thesis investigates how to express and reason about relational concepts and methods inside the constructive logical framework of Martin-Löf's monomorphic type theory. We cover several areas where the notion of relation is central, and show how to formalize the basic concepts of each area. This formalization is carried out in a computer proof assistant for Martin-Löf's type theory called Agda/Alfa, and this software allows the practical handling of the numerous technical details involved in such a task.

The first area we investigate is relations and relational systems themselves. Many of the relational systems in use within computer science are based on algebras that extend Boolean algebra, and hence are unsuitable for a constructive framework. Instead we use the constructive algebra of relations provided by the categorical theory of allegories. First, we introduce the notion of an E-allegory, the appropriate abstract notion of an allegory formalizable in Martin-Löf type theory. Then we set up two concrete instances: the E-allegory of E-relations, and the E-allegory of finite decidable E-relations. We investigate further properties of the E-allegories and note that only the latter satisfies the axioms of a power allegory and a Boolean allegory.

The second area is relational program construction and derivation. Martin-Löf type theory provides its own approach to program construction via the identification of propositions and types, and our goal is to supplement this methodology with techniques from relational programming. Here we note that Martin-Löf type theory is based on a functional notion of program, and provides a logical setting where we can reason about the connections between functional and relational notions of programs. We show some links and differences with type-theoretic program construction and derivation. We then show how recursive relations can be dealt with in type theory using the concept of relational catamorphism.

The third area is the relational database model. This data model has similarities and connections with relational algebras and allegories, but presents new problems in the way typings are handled via schemes and attributes. We set up an increasingly complex series of formalizations of the relational model, with different decidability characteristics and with respect to the way in which the scheme/attribute typing issues are handled. The final formalization, which is both the most complex and the most satisfactory, is built on an abstract finite set layer, and can be seen as an abstraction for implementations of database managers.

Keywords: formalized mathematics, relational systems, category theory, programming logics, constructive type theory, logical frameworks, relational database model.

ACM Classification: F.3.1 [Specifying and Verifying and Reasoning about Programs]: Logics of programs, Mechanical verification; H.2.1 [Database Management]: Logical Design; F.4.1 [Mathematical Logic]: Lambda calculus and related systems.

AMS MSC: 18B10 (Category of relations); 68P15 (Database theory); 68Q60 (Spec-

ification and verification: program logics); 03B20 (Subsystems of classical logic: intuitionistic logic); 03B35 (Mechanization of proofs and logical operations).

Acknowledgements

First and foremost, I want to thank my supervisor, Peter Dybjer. This work would not have been born (much less seen completion!) without his guidance, support, and seemingly infinite amounts of patience. He has been a big inspiration, a source of knowledge, and an essential help every time I got stuck in some difficult spots during this work.

Some people provided invaluable assistance during many parts of this work. Makoto Takeyama was always there to help with my doubts about a lot of technical details, and the same is true of Michael Hedberg. Mary Sheeran and Catarina Coquand, who make up the rest of my PhD advisory commitee, have also offered many comments and advice about this work. My old officemate, Qiao Haiyan, was also a great aid for parts of this work, not to mention dealing with my odd habits with a remarkably good sense of humour (and this last should also be said of my other officemates along the years!).

Thanks are due to the Programming Logic Group at Chalmers for funding, feedback, and for providing a stimulating environment for my work. I also want to thank everybody at the Department of Computing Science here at Chalmers and Göteborg Universities for their funding, and for providing a comfortable and very enjoyable work environment, both technically and human-wise.

During the first 3 years of my studies and research, I was fully funded by the FOMEC project of the Departmento de Ciencias de la Computación, Universidad Nacional del Sur, Bahía Blanca, Argentina. My debt of gratitude with the Department's people goes beyond words. I particularly need to acknowledge Guillermo R. Simari, Pablo R. Fillottrani and Juan C. Augusto for their unfailing help with the many practical details during the course of my FOMEC grant. Everybody else at DCC-UNS, my thanks to you too and do feel included here, even if you are not explicitly mentioned.

On two occasions, my attendance at the RelMiCS seminars was helped by funding from the organizers, for which I am very thankful. Many of the participants offered valuable comments and pointers about the subject of this work.

Last but far from least, my special thanks to my family and friends, wherever else in this strange world you are.

Carlos Gonzalía

Göteborg, January 2006.

Contents

Chapter 1

Introduction

1.1 Scope of this work

We investigate how relational concepts and methods can be described, formalized, and made use of inside Martin-Löf's type theory [54]. A goal of this work is to construct a basic but complete formalization of relations and their properties in a type-theoretical logical framework. This formalization will be of a constructive nature, and make heavy use of dependent types. Three areas are of particular importance in this regard, and we concentrate our work on them. We start with relations themselves as mathematical objects, and their properties when defined in type theory. Then, we consider the use of relations for describing and expressing programs, that is relational calculi for program specification and derivation, and the question of how to deal with relational versus functional programs in type theory. Finally, we deal with relational database theory, and the possibility of specifying and reasoning about implementations of relational database systems in type theory. All this work is carried out with the help of the computer proof assistant Alfa/Agda [18, 36] that implements a type-theoretic logical framework, ensuring a rigorous and error-free handling of the many details involved.

1.2 Purpose and goals of this work

Why have we embarked on this investigation? The answer has several parts, depending on the point of view we take: Martin-Löf's type theory, relational programming, or the relational database model.

1.2.1 The point of view of type theory

In type theory, there is a general programme of formalizing constructive mathematics and programming using its framework of propositions-as-types and its restricted functional language (all of whose functions terminate). Within this programme, not much work has been done on the subject of relation calculus. Hence any detailed exploration of that subject starting from its basic notions is worthwhile.

A first goal is to set up the constructive formal algebra of relations that corresponds to the existing set-theoretical one (of categories of relations called allegories). We achieve this with a formalization that closely follows the different kinds of abstract allegories.

A second goal is to show that the "intended models" of those abstract constructive allegories can be set up in a corresponding way to the set-theoretic ones. Again, we achieve this with a formalization of constructive categories of concrete relations in which equality is handled within type theory in an explicit way. Our intended model will however not be a power allegory or a boolean allegory. Hence the "intended models" for those kinds of abstract constructive allegories need to be restricted. We show that a sufficient restriction is to have concrete relations that are decidable and act over finite domains and codomains.

1.2.2 The point of view of relational programming

We can use type theory for formulating a variant of the relational programming techniques developed by e.g. Bird and de Moor [10]. This would allow us to construct programs from specifications using both relational methods and type-theoretic ones. One difference is that we use constructive logic and the constructive allegories mentioned above. Another difference is that the functional language provided by type theory is different from the standard ones such as Haskell, and in particular there is a need to deal explicitly with termination issues.

The goal is to show that we can use the relational concepts provided by our already mentioned formalization of constructive allegories, and carry out relational program calculation inside the framework of type theory. A central part of such calculations involves datatypes, expressed using functors, initial algebras, and their morphisms. We show that relational programs can be expressed using our formalization, extended with notions that correspond to those used in set-theoretic formalizations for dealing with datatypes. We also show that we can reason about properties of such relational programs.

We then discuss by means of example how we can avoid the use of power constructions. Relational power operations, while used to great advantage in the literature on relational program derivation, are as we mentioned before not available within the type theory framework we use. A more detailed exploration of this issue would be part of our future work.

We finally remark that all of this effort is backed by the mechanical support provided by the proof assistant, making it both easier to be sure of its correctness, and allowing its use as a library for relational concepts and programs in type theory.

1.2.3 The point of view of relational databases

Finally, we consider the relational database model within the programme of constructive formalization inside type theory. This model is related but somewhat different to relation calculus (see [70] for details), but it hasn't been much investigated either as part of that programme.

A first goal is to adequately capture the abstract concepts of the model, both at the level of data components (attributes, schemes) and the relational database operations which are used for retrieving meaningful information from an existing database. We achieve this by showing successive, ever more complex, concrete formalizations of the model. This makes evident the trade-off involved in such an effort: capturing more and more precisely the notions of scheme and attributes makes the concrete formalizations more and more difficult to build upon or prove things about. A solution is then provided by formalizing a completely abstract definition of the relational model based on abstract finite sets and their operations.

A second goal is to validate concrete implementations of databases by proving that they satisfy the axioms of the abstract definition we provided before. We do this for a simple case, in which the underlying finite sets are implemented by means of finite lists without repeated elements. Once again, it is important to remark that the mechanical support provided by the proof assistant allows us to correctly carry out the validation of an implementation and also the specification of the relational database model.

1.3 A guide to the rest of this work

1.3.1 Relational systems

Relations have a long history as a subject of investigation, where Tarski is the first modern proponent of their study and their use as a tool in algebraic logic [67]. Relational methods for computer science, which originated in algebraic logic, have proven their success and interest since at least 1970. A good survey of both their foundations and many of the areas in which they are applied can be found in [12]. A difficulty one finds when attempting to carry out a programme of the kind described in the previous section is that the relational systems used are typically extensions of Boolean algebras, and hence unsuitable for an intuitionistic logical framework. An alternative approach, developed somewhat in parallel with the algebraic logic one, has been based on categories of relations, called allegories

[31]. In the first part of the thesis we show formally that a variant of the notion of an allegory provides a suitable abstract framework for relations inside Martin-Löf type theory. Category theory has been investigated in type theory quite extensively [39, 41], so in our efforts we can resort to ideas and techniques from formalizations and previous work.

1.3.2 Allegories and programs

An allegory is a category with some extra structure. The arrows in an allegory can be thought of as abstract relations. A good amount of work is available in the literature about them, starting with the foundational work of Freyd and Scedrov [31]. Their use as a formalism in which to specify, derive and reason about programs is due to many different researchers, but the work of Bird and de Moor [10] provides the most systematic and expositive presentation. We base our investigation on it, and define type-theoretic concepts corresponding to the several kinds of allegories used in practice. We then show that we can create allegories of type-theoretical relations with the desired properties. However, we note in particular that we will in general not have a power allegory: for example, the E-allegory of E-relations, the constructive analogue of the allegory of relations, is not a power allegory. We then proceed to show that we can create allegories of type-theoretical relations with these notions by restricting the kind of relations allowed: we require the sets over which the relations hold to be finite and the relations to be decidable. After that, we deal with the problem of defining relations by recursion in type theory, and how to represent and reason about programs based on them inside our formalization.

1.3.3 Relational databases

The relational model of databases was introduced by Codd [17] more than 30 years ago, and has been hugely successful. Its theoretical underpinnings are somewhat different than those of relational algebras or allegories (though a formal connection can be made: for instance [57] for allegories and databases, [70] for Tarski's systems and databases, [58] for specially devised algebraic logic to deal with databases), in that relations can have arbitrary arities rather than the binary ones of those other two areas. Questions of typing also become more convoluted, with the presence of global labels and typings for them. Hence we investigate relational database concepts in a separate approach, and set up formalizations of growing complexity while trying to capture those questions of typings and labels in a way that works in type theory. The final formalization and the more rewarding one uses an abstract layer for finite set data structures. This allows us to show how implementations of relational databases rely principally on the implementation of abstract finite sets. To illustrate the details involved in proving that such an implementation satisfies the abstract behaviour, we provide a detailed proof for lists without repetitions as the implementing data structure.

1.3.4 Publications

Some of the main ideas in this thesis were summarized in the two following two publications:

- C. Gonzalía: Towards a Formalisation of Relational Database Theory in Constructive Type Theory. In R. Berghammer, B. Möller and G. Struth (eds.): *Relational and Kleene-Algebraic Methods in Computer Science*, 7th International Seminar on Relational Methods in Computer Science and 2nd International Workshop on Applications of Kleene Algebra, Bad Malente, Germany, May 2003, Revised Selected Papers. LNCS 3051, Springer-Verlag, 2004., pp. 137–148.
- C. Gonzalía: The Allegory of E-Relations in Constructive Type Theory. In J. Desharnais, M. Frappier and W. MacCaull (eds.), *Relational Methods in Computer Science - The Québec Seminar*, pp. 19–38. Proceedings Series, Vol. 1. Methoδos Verlag, 2002.

An early version of the material about the formalization of E-allegories in type theory appeared as our licenciate thesis:

- C. Gonzalía: *Relation Calculus in Martin-Löf Type Theory*. Licenciate thesis. Technical Report no. 5L, Department of Computing Science and Engineering, Chalmers University of Technology and Göteborg University, 2002.

Apart from minor changes in the details of the formalization and a much different presentation of it, the main difference resides in the finite allegory part: while in the licenciate work we built on an abstract layer for finite predicate logic whose implementation was left pending, here we have implemented almost everything required for the definitions and proofs about finite decidable allegories.

1.3.5 Related work

Theorem provers for relational systems

There exist several theorem provers for relation calculus. aRa [65] is an automatic theorem prover for various kinds of relation algebras. It works by converting relational equations to finite variable first-order logic sentences. Dawson and Goré [22, 23] have implemented the Display Logic calculus for relation algebra as an Isabelle theory. (Display Logic is a syntactical proof system for non-classical logics.) It can be used as an interactive proof assistant for relation algebras, and provides some automatisation of tedious tasks. RALL [55] is a theorem proving system for relation algebra, based on Isabelle. It is based on a HOL theory that is a standard Isabelle object logic, and is heavily dependent on atomicity properties of the relation algebra being used. MSPASS [42] is a theorem prover for (among other logics) relational calculus, based on translation to first-order logic with equality. Formisano, Omodeo and Temperini [30] have worked with the theorem prover Otter to automatise Tarski-Givant's map calculus, an equational language for relations. RelDT [20] is a relational theorem prover based on advanced tableaux methods, and it has an extension called RelDT-DB that can deal with relational database formulas. The core of its method comes from a previous relational theorem prover, ReVAT [49], which like RelDT is implemented in Prolog. Finally, Kahl [44] has worked on calculational proofs of relation algebra formulas using the Isabelle system.

An obvious difference of all the above mentioned work with respect to ours is of course that we use a proof assistant and not a theorem prover, and also our aim is the formalisation of a body of mathematical notions and properties of relational systems instead of having representations suitable for automatic proof search. A less apparent difference but an essential one, is that all those works (with the exception of Kahl's) deal with the classical relation calculus in one way or another, that is, as formal systems that extend a Boolean algebra.

Relational formula manipulation systems

There exist some software tools to deal with manipulations of relations or relational formulas. We already mentioned Dawson and Goré's system, and Kahl's calculational theories for Isabelle. This dual nature of those two works comes from the fact that Isabelle is typically used both as a purely automatic prover, and also as an interactive proof assistant. RALF [37] is a system and proof assistant for graphically manipulating relation algebra formulas (by applying rules to the expression graph of the formulas), and as such a pretty unique example of its kind. Its transformation rules are hard-coded and do not form a formal calculus, however. Our work is similar in nature to those two built on top of Isabelle, but much more extensive.

Concrete-relation computation systems

RELVIEW [7] is a system for prototyping relational specifications and programs. It provides relations as Boolean matrices with their operations, and allows the user to create programs that have such relations as data. The language is imperative, offers an API for external calls, and the actual implementation of the relations is by means of special binary decision diagrams. RATH [46] is a library of Haskell modules designed to allow the user to construct and test concrete relations for properties. All the standard kinds of relational category definitions are available for such tests. Another language, Grok [40], is based on binary relations and their operations, and its main use is as a tool for analyzing the architectural structure of software. It is imperative, and its conceptual model for relations is as lists of tuples.

Finally, we must mention the few efforts towards a possible relational programming language paradigm (not to be confused with the logic programming one). In this area, Cattrall has been the most ambitious effort to date, with the language Drusilla [16], which had laziness and a complex type inference system. A different kind of effort is Dwyer's Libra [24], untyped and which executes in a way similar to Prolog.

With regard to our work, while the Agda/Alfa system provides a simple way of computing an expression, it is not intended as a feature for evaluating big and complex ones, or in any efficient way. New features in the future version of this system could make such computations more practical, though.

Relational hardware specification languages

In a related field which we haven't explored, relational notations for hardware specification and reasoning about circuits have been used with success [13]. In particular, a subset of the design language Ruby [43] has been formalized by Rasmussen [62] in Zermelo-Frankel set theory using the Isabelle theorem prover. Vaccari [68] has used the proof checker PVS to verify his calculational style of Ruby, and to develop his case studies with that tool support. Since there is a clear connection between relational hardware specifications and allegories, as shown by [13], our work intersects the above mentioned ones. While the exact kind of allegories corresponding to Ruby are not present in our work, there is seemingly little distance to cover if we would want to formalise them as an addition to our work.

Relational database model formalizations

A thorough formalisation of data collections in Martin-Löf's type theory has been carried out by Rajagopalan and Tsang [60, 61]. This allows them to deal with data models, abstract representations of database systems. The resulting algebra is then more general than a relational one, and other semantic data models are also representable in terms of their generic algebra. This formalisation was carried out in Nuprl, allowing for extracted functions that can be executed and serve as prototypes for database manipulation operations. The result is somewhat comparable to our final, abstract formalization based on an abstract finite set datatype axiomatization. However, our aim was to directly capture the relational model, so no such generic algebra of data collections is present. We use a concrete implementation of collections (inspired by their efforts) when showing an example of an implementation of databases satisfying our abstract formalization.

In a different vein, the work by Barros [4] deals with the derivation of programs that implement a specification of a particular database, that is, the particular semantic structure of the data the database is required to store. Finally, a formal calculus that can be used for automatically proving properties of relational database theory has been developed by MacCaull and Orlowska [50]. The theorem prover RelDT-DB mentioned previously is intended for use with this formal system for relational databases.

1.3.6 Further work

The resulting formalizations are available to other users of the computer proof-assistant, and are consistent with the current standard libraries for it. Additional work would be to extend the list of derived properties satisfied by the different kinds of allegories, so as to provide a more extensive set of laws to use for relational program manipulation and performing proofs of program properties. A different possibility is to extend the formalization with new kinds of categories of relations, for instance Kleene categories (and their variants with domains and with tests) or Dedekind categories. Doing so would probably need a refactoring of the hierarchy of definitions for the whole set of definitions. For details of such an approach and also definitions of these other categories of relations, see [45].

Another, more difficult but probably more rewarding programme to follow would be to formalize some more complex examples of Bird and de Moor's relational program derivation inside type theory, in particular ones dealing with the richer ways of expressing greedy and dynamic programming strategies in their work. This would require additional formalizations of some concepts used in those strategies, at the same time as we must be careful to avoid the explicit use of general power relations, which are not available in the framework of type theory.

An additional programme would be to formalize and prove correct some more realistic data structure used in the implementation of relational database managers. Concrete systems are typically based on optimized kinds of trees or hash structures, which may require a substantial formalization effort. Therefore, it might be wise to first formalize a simpler but still efficient data structure such as some version of balanced search trees.

Yet another separate possibility would be to attempt a formalization of the E-R (entity-relationship) data model. This one is a semantic kind of model in which the structure of data sets (as sets of tuples) and the relationship of such sets to each other (as a kind of mathematical relations with participation constraints) is represented in the shape of a graph.

Chapter 2

Dependent Type Theory and the Proof Assistant Alfa

We will here give a brief introduction to Martin-Löf's type theory (MLTT) and the Alfa system. While explaining the notation and features of Alfa [36] we will also go through some of the main features of MLTT. For a comprehensive survey of MLTT, see [54].

2.1 Type theory

The formal system in which we work is a descendant of Martin-Löf's monomorphic type theory (MLTT) [54]. MLTT is a framework based on the lambda calculus with dependent types extended with inductive definitions, which correspond roughly speaking to certain kinds of "well-formed" recursive data types. It is formulated in natural deduction style and allows us to work with intuitionistic predicate logic via the Curry-Howard isomorphism. Type theory provides a setting in which we can formalize constructive mathematics, but also a functional programming language with dependent recursive types where all programs terminate. It is also good for specifying and extracting programs, as we can consider that a program's specification is the same as a proposition/set, and the actual program is the proof/element of that proposition/set.

MLTT was at first primarily developed as an attempt to clarify the syntax and semantics of constructive mathematics. It has its own meaning theory where the semantics is usually given in a direct way through intuitive explanations of the basic concepts. For instance, the central notion of set is explained by saying that a set is defined by stipulating how its canonical elements are formed and when canonical elements are equal. The theory is open in that we can introduce new sets by inductive and inductive-recursive definitions, in ways described for instance in [25, 26]. In doing so, possibly dependent inductive types (also called inductive families) can be created, and we can define functions by structural recursion.

2.2 Alfa

Alfa [36] is a graphical proof editor for MLTT based on the proof checker Agda [18], which in turn is based on the manipulation of explicit proof objects in the setting of MLTT. The formal syntax and semantics of the Agda language is what has been called *structured type theory* [19], a version of type theory where we have dependently typed products with labeled fields (this, basically a dependently typed record, is called a signature and a value of such a type is called a structure), algebraic data types, local definitions and a package mechanism. The concrete syntax of Agda resembles Haskell's, and is based on Cayenne [2] (a dependently typed functional programming language).

On top of these features, the Alfa editor allows the user a lot of flexibility to define how proof terms are presented on the screen. Fundamental to the use of Alfa as an interactive proof assistant is how it allows us to leave some parts of our proofs incomplete, by using a notion of *meta-variable* with a well-defined behaviour. These meta-variables can then be filled in by the user, with the Agda engine checking the correctness of all introduced or refined expressions. The meta-variables are denoted by $?_n$, and can appear in any place inside an expression, where a subexpression is not known. For instance, an application of a unary function f to an argument which is not yet known temporarily appears as $f\ ?_n$ during the interactive development of the piece of proof code where that application is used.

We remark that currently there is a new standard library for Alfa, developed by Michael Hedberg [38], which we make abundant use of. We usually mention the main typings of the components of this library that we use, but for reasons of space we suggest the interested reader to browse through the actual library for the technical details about many long definitions.

2.3 Alfa's notation

MLTT is a typed lambda calculus with dependent types. We will write $a \in A$ to mean that the term a has type A (this is called a judgement in type theory, and is also written as $a : A$ in much of the literature). There are function types, written $f \in A \to B$, expressing that f is a function with domain A and codomain B. Also, there are dependent function types, written $f \in (x \in A) \to B$, expressing that the codomain B can depend on the value of x in A.

Functions can be defined using lambda notation, with the arguments and the body of the lambda expression separated by an "$\to$" (Alfa uses this instead of the usual dot). For example, we can define a generalized identity function *id*, where the first argument is a set A such that when we apply *id* to A we get the identity function $id\ A \in A \rightarrow A$ on A:

$$id \in (A \in Set) \rightarrow A \rightarrow A$$

$$id \equiv \lambda A\, x \rightarrow x$$

Definitions consist of a header and a body. The header includes the name of the concept we are defining and lists its parameters (along with their types), followed by its type after a $\in$ sign. The body repeats the header's left side in an abbreviated form, and gives the content of the definition after an "$\equiv$" sign. On many occasions it will be convenient to use argument hiding, which causes some arguments to become implicit and invisible in an application of the defined function, while in the actual definition they will show up as having been moved from their position on the header to the right of it, in a special group of parentheses.

Continuing with the identity function example, in Alfa we can also write:

$$id(A \in Set) \in A \rightarrow A$$

$$id\ A \equiv \lambda x \rightarrow x$$

and the meaning is the same. Or as a third alternative we could move the x too to the left of the equivalence if we prefer the look of the following definition:

$$id(A \in Set, x \in A) \in A$$

$$id\ A\ x \equiv x$$

Argument hiding is a presentation feature of Alfa: it doesn't alter the proof code or its meaning, and it just changes the way some expressions will look on the screen. The hiding of some arguments can greatly enhance readability, by removing arguments that typically only show detailed typing information that is not really essential to the understanding and use of the function in question. For instance, hiding the argument A of the identity function would give:

$$id \in A \rightarrow A \qquad (A \in Set)$$

$$id \equiv \lambda x \rightarrow x$$

The hidden argument appears on the right of the typing declaration, within parentheses. Alfa signals that a definition has hidden arguments by using this special layout. Using this hiding feature we can make definitions look polymorphic as in Haskell.

Alfa also allows other presentation enhancements of the proof code on the screen, such as replacing the names of defined functions by mathematical symbols or new aliases, making such names infix, giving them priority and associativity information, and more. We will make use of several of these features, and refer the reader to the Alfa homepage for the details [36].

A dependent record, called a signature, is a kind of dependent product where each component is labeled. We can have an arbitrary finite number of fields. Dependent records are denoted by expressions beginning with the keyword **sig**. The fields are listed after it with their respective types. A value of such a record type, called a structure, is denoted by an expression beginning with the keyword **struct**. The values of the fields are listed after it, with the labels followed by an "≡" and their particular value. Accessing individual fields can be done with dot notation.

For instance, here is a signature definition which declares the type of dependent pairs with the first component $fst \in A$ and the second component $snd \in B\,x$:

$Sum\ (A \in Set, B \in A \to Set) \in Set$

$Sum \ \equiv\ \mathbf{sig}\ \{fst\ \in A;\ \ snd\ \in B\,x\}$

and a corresponding structure or instance of that type (assuming we have a pair of values $a \in A$ and $b \in B\,a$):

$s \in Sum\,A\,B$

$s\ \equiv\ \mathbf{struct}\ \{fst\ \equiv a;\ \ snd\ \equiv b\}$

This is Alfa's notation for a Σ type, often denoted $\Sigma x \in A.B\,x$. We can intuitively read the above definition as "$Sum\ A\ B$ is the sum of all $B\ x$ for $x \in A$".

For this example, we can refer to $s.fst$ for instance, and this would be an element of A. Note that dependencies on the types of the fields can be introduced by just using field names in other field's typings. Thus, we have in Sum that the type of snd mentions fst and so depends on its value. Also, note that $\mathbf{struct}\ \{fst\ \equiv a;\ \ snd\ \equiv b\}$ is just a notation for the pair (a, b) where we have the additional information that the first component can be accessed by the label fst and the second by the label snd using dot notation. Regarding notational convenience, just like in Haskell, nestings and levels can be indicated by layout and spacing conventions.

Data types are denoted by the **data** keyword. The constructors and their arguments are listed after it, in the style of Haskell syntax. This is the same as an inductive definition of a set. For information about exactly what inductive definitions are admissible in MLTT, see [26], and for an explanation of the more general principles for inductive-recursive definitions see [25]. In the present work we will not use any inductive-recursive definitions except the usual universes.

As an example, consider the set of Booleans with constructors **true** and **false**, taken from the Alfa library:

$Bool\ \in Set$

$$Bool \equiv \mathbf{data} \left\{ \begin{array}{l} \mathbf{true} \\ \mathbf{false} \end{array} \right\}$$

A case expression, denoted by the keyword **case**, allows pattern matching on values of such data types. However, Alfa automatically presents definitions whose parameters are on the left hand side of the $\in$ in the typing header as if they were made by a series of equations where the arguments are pattern-matched, following Haskell conventions. As an example of this automatic pretty-printing, we take the definition of Boolean disjunction from the Alfa library. This will also illustrate the use of the infix and symbol representation features of the Alfa system (both are picked by the user from a special menu available for defined names, and are not indicated on the proof code by any special keywords or visible annotations about fixity or symbols, as might be used in other systems):

$$\vee\ (x, y\ \in\ Bool) \in Bool$$

$$\mathbf{true}\ \vee\ y \equiv \mathbf{true}$$

$$\mathbf{false}\ \vee\ y \equiv y$$

However, if the arguments on which the definition proceeds by pattern-matching are on the right hand side of the $\in$ in the typing header, then **case** will appear, as this modified Boolean disjunction definition shows:

$$\vee\ \in Bool\ \rightarrow\ Bool\ \rightarrow\ Bool$$

$$\vee \equiv \lambda\, x\, y\ \rightarrow\ \mathbf{case}\ x\ \mathbf{of} \left\{ \begin{array}{l} \mathbf{true}\ \rightarrow\ \mathbf{true} \\ \mathbf{false}\ \rightarrow\ y \end{array} \right\}$$

Definitions can be recursive and so the question arises about when they are correct. MLTT only allows terminating functions, by using just primitive (or structural) recursion on the elements of a data type (see for instance [28]). In Agda/Alfa a more general principle of "structurally smaller recursive calls" is allowed, and the proof assistant offers a simple termination checker that the user should manually invoke on his or her definitions, to make sure they comply with this recursion principle.

It is also possible to introduce local definitions, which follow the same form as in Haskell, with **let** ... **in** ... syntax. As an example, consider the following one (admittedly quite artificial, and built so just **let** can be illustrated), defining Boolean and in terms of the previous or and a local definition for Boolean not:

$$\wedge\ (x, y\ \in\ Bool) \in Bool$$

$$x\ \wedge\ y \equiv \begin{array}{l} \mathbf{let} \left[\begin{array}{l} \neg \in Bool\ \rightarrow\ Bool \\ \neg\ \equiv\ \lambda\, x\ \rightarrow\ \mathbf{case}\ x\ \mathbf{of} \left\{ \begin{array}{l} \mathbf{true}\ \rightarrow\ \mathbf{false} \\ \mathbf{false}\ \rightarrow\ \mathbf{true} \end{array} \right\} \end{array} \right. \\ \mathbf{in}\, \neg\, (\neg\, x\ \vee\ \neg\, y) \end{array}$$

As we already have seen, we have a special type *Set* of "sets" in MLTT, and in Agda/Alfa. It is the first universe in a hierarchy of "universes", and is also called #0. The next universe is *Type*, and is also called #1. Higher universes are denoted by #2, #3, ... It is always the case that $\#n \in \#(n+1)$.

2.4 Predicate logic

Propositions are interpreted as sets by the Curry-Howard isomorphism. Here we introduce the type *Prop* of propositions:

$Prop \in Type$

$Prop \equiv Set$

We introduce (following the Alfa library) as a convenient shorthand the type *Pred X* of predicates on a given set X. Predicates are nothing but propositional functions, i.e.:

$Pred\ (X \in Set) \in Type$

$Pred\ X \equiv X \rightarrow Prop$

Observe that $Pred\ (X \in Set) \in Type$ is just another notation for $Pred \in Set \rightarrow Type$.

The types for the logical constants are as follows:

$\supset \in (X, Y \in Prop) \rightarrow Prop$

$\perp \in Prop$

$\top \in Prop$

$\neg \in (X \in Prop) \rightarrow Prop$

$\& \in (X, Y \in Prop) \rightarrow Prop$

$\leftrightarrow \in (X, Y \in Prop) \rightarrow Prop$

$\vee \in (X, Y \in Prop) \rightarrow Prop$

$\forall \in (X \in Set, P \in Pred\ X) \rightarrow Prop$

$\exists \in (X \in Set, P \in Pred\ X) \rightarrow Prop$

For $\top$, the true proposition, we should remark that it is a set consisting of a single element. Following the Alfa library convention, we denote that element by **tt**.

As an example of the definitions of the logical constants in the Alfa library, consider disjunction:

$\vee\ (X, Y \in Set) \in Set$

$X \vee Y \equiv Plus\, X\, Y$

where *Plus* is the disjoint sum datatype with constructors **inl** and **inr**:

$Plus\ (X, Y \in Set) \in Set$

$$Plus\, X\, Y \equiv \mathbf{data} \left\{ \begin{array}{l} \mathbf{inl}\,(x \in X) \\ \mathbf{inr}\,(y \in Y) \end{array} \right\}$$

2.5 Booleans

Booleans were defined in the preceding section, here we will deal with some definitions using them.

2.5.1 Some useful definitions

A decidable Boolean predicate on a set X is simply a Boolean unary function:

$pred\ (X \in Set) \in Set$

$pred\, X \equiv X \rightarrow Bool$

To be able to assert that a certain Boolean has the value **true**, we define a function which maps a Boolean value p to the corresponding proposition $|p|$:

$|\_|\ (p \in Bool) \in Prop$

$|\,\mathbf{true}\,| \equiv \top$

$|\,\mathbf{false}\,| \equiv \bot$

When using such $|p|$ proof objects, we usually speak of having "lifted" the boolean value.

Note that we have shown two definitions of disjunction, one Boolean and the other propositional. The use of this last function $|\_|$ allows one to move from the Boolean versions to the propositional ones. The Alfa library provides some useful lemmas relating to such transformations.

2.5.2 Operators

The definition of unary and binary operators in general is given by:

$Unop\ (X \in Set) \in Set$

$Unop\, X \equiv X \rightarrow X$

$Binop\ (X \in Set) \in Set$

$Binop\, X \equiv X \rightarrow X \rightarrow X$

The types and notation for the Boolean operators, for future reference in other chapters, are as follows:

$\top \in Bool$

$\bot \in Bool$

$\neg \in Unop\, Bool$

$\vee \in Binop\, Bool$

$\wedge \in Binop\, Bool$

$\Rightarrow \in Binop\, Bool$

$\Leftrightarrow \in Binop\, Bool$

Note that we overload notation and use the same symbols for Boolean operators as for propositional ones.

2.6 Natural numbers

The natural numbers are defined as an algebraic data type:

$Nat \in Set$

$$Nat \equiv \mathbf{data} \left\{ \begin{array}{l} \mathbf{zer} \\ \mathbf{suc}\,(m \in Nat) \end{array} \right\}$$

As an example of a function involving natural numbers, and also one that uses (correct) recursion, consider the sum operation from the Alfa library:

$+\ (m, n \in Nat) \in Nat$

$\mathbf{zer} + n \equiv n$

$\mathbf{suc}\, m' + n \equiv \mathbf{suc}\ (m' + n)$

2.7 Finite sets

A notion of finite set is provided by the library. Intuitively, such a set with k elements can be thought of being $\{0, 1, \ldots, k-1\}$. It is denoted N_k, or following the library, *Fin k*.

$Fin\ (k\ \in\ Nat) \in Set$

$Fin\ \mathbf{zer} \equiv Zero$

$Fin\ (\mathbf{suc}\, k') \equiv Succ\ (Fin\, k')$

Note that $Fin\, k$ is a dependent type, it depends on $k \in Nat$. It is defined by recursion on k. Also note that it is possible to define a set by primitive recursion in Alfa.

In the above definition, *Zero* is the empty set. It is defined as the set generated by no constructors:

$Zero\ \in Set$

$Zero \equiv \mathbf{data}\ \{\}$

and *Succ* is a set forming function that extends a set by adding to it a new element:

$Succ\ (X\ \in\ Set) \in Set$

$$Succ\, X \equiv \mathbf{data} \left\{ \begin{array}{l} \mathbf{zer} \\ \mathbf{suc}\,(x\ \in\ X) \end{array} \right\}$$

Elements of *Fin k* have a similar representation to naturals, that is in a unary-style notation. While the constructors look the same in the presented proof code, they have different types which are handled correctly by Alfa.

2.8 Finite decidable quantifiers

To deal with situations in which decidable quantification is wanted, we will resort to quantifiers acting on domains of the form *Fin m*:

$\exists \in (m\ \in\ Nat) \rightarrow pred\ (Fin\, m)\ \rightarrow Bool$

$\forall \in (m\ \in\ Nat) \rightarrow pred\ (Fin\, m)\ \rightarrow Bool$

Their definitions are part of the standard Alfa library, as are several of their properties that we'll have use for in some of the following chapters. For their meaning, consider the existential quantifier: $\exists\, m\, p\ =\ \mathbf{true}$ iff there exists an $n\ \in\ Fin\ m$ such that $p\, n = \mathbf{true}$. The implementation of both quantifiers is by induction on m. As an illustration, we show a rewritten version that basically captures the way the finite existential quantifier is provided by the Alfa library:

$\exists\ \in (m\ \in\ Nat, f\ \in\ Fin\, m\ \rightarrow\ Bool) \rightarrow\ Bool$

$$\exists \equiv \lambda\, m f\ \rightarrow\ \mathbf{case}\ m\ \mathbf{of} \left\{ \begin{array}{l} \mathbf{zer}\ \rightarrow\ \mathbf{false} \\ \mathbf{suc}\, m'\ \rightarrow f\,\mathbf{zer}\ \vee\ \exists\, m'\ (tail\, m' f) \end{array} \right\}$$

where *tail* adapts a predicate over *Fin* (**suc** *m*) to make it work over *Fin m*:

$tail\ (m \in Nat, f \in pred\ (Fin\ (\mathbf{suc}\, m))) \in pred\ (Fin\, m)$

$tail\, m\, f \equiv \lambda\, n\ \rightarrow f\ (\mathbf{suc}\, n)$

The definition of the finite existential quantifier above is a valid recursive definition of Alfa. As already mentioned, the Alfa system includes a simple termination checker that the user can apply to some definition to make sure it follows an allowed form of recursion.

Chapter 3

Relations in Type Theory

This chapter addresses the question of how relations can be represented in type theory and will introduce many issues that will appear again in the rest of this work. In that regard, it will be important to point out the differences with how relations appear and are used in naive set theory, programming languages and data structures.

3.1 Relations in set theory and in type theory

In set theory, the usual way to define a relation R is by stating $R \subseteq A \times B$, that is, it is a subset of a cartesian product. However, if X and Y are sets in the sense of type theory it is not meaningful to ask if $X \subseteq Y$, since it is not a judgement in type theory. (There are proposals for extending type theory with subtype judgements, but this is not supported by Agda/Alfa.) The question might arise in the reader's mind of why we can't define relations as subsets but using "$\in$" typing judgements. But why does it not work to implement $X \subseteq Y$ as $x \in X \vdash x \in Y$? The reason is that type theory is monomorphic, so x cannot have two different types X and Y.

An alternative (and equivalent) way of dealing with membership in classical set theory is by using characteristic functions. Thus, for a subset $X \subseteq Y$ we would have a function $f_X : Y \to \mathit{Bool}$ such that $f\,x = \mathbf{true}$ iff $x \in X$. This would mean that instead of having a relation $R \subseteq A \times B$, we can define a function $\phi_R : A \times B \to \mathit{Bool}$ such that $\phi_R(a, b) = \mathbf{true}$ iff $a\,R\,b$.

This last definition can also be used in type theory, but it has a different meaning, since $\phi_R : A \times B \to \mathit{Bool}$ is a *computable* function in type theory. Since we want to consider relations which are not decidable, however, we can instead consider $\phi_R : A \times B \to \mathit{Set}$ so that a and b are related by R iff $\phi_R(a, b)$ is inhabited. Additionally, to enjoy the convenience of curried functions in such a situation, we can instead use $\phi_R : A \to B \to Set$ (or *Prop* instead of *Set*, as they are the same in this setting) as a good type theory representation.

3.2 Relations in programming languages

Another relevant issue is how relations are represented in programming. Programming languages usually lack a relation datatype as a feature. Instead, implementations of finite relations are often provided by a library for the language, with an abstract interface to some concrete representation and implementation for the operations. Since relations can also be seen as directed graphs, standard representation techniques for graphs can sometimes be used, for instance adjacency lists and adjacency matrices. There also exist some advanced representation techniques, for instance the special binary decision diagrams used in Relview [7], which is also an exceptional case of an imperative programming language with a built-in relation datatype.

It is useful to note that a finite binary relation can be seen as a special case of a dictionary. We can define a dictionary as a collection of pairs (k, e) consisting of a key value and an element value, such that for any value of k there is at most one such pair in the dictionary (in some data structures textbooks, this is called a map, and dictionaries allow repeated pairs with the same value of k). A finite relation $R \subseteq A \times B$ can then be a dictionary in which the keys come from A and the elements are lists (or some other data structure representation for a finite set) of values of B. That is, the dictionary is made up of pairs $(x, \{y \mid x\,R\,y\})$. In this way, finite relations can be implemented via any of the standard implementation techniques for dictionaries, for instance as association lists or hash tables.

While the preceding discussion applies to most programming languages regardless of their particular features, there are features present in non-imperative languages that allow other ways of expressing relations. In lazy functional languages one can use infinite lists for representing infinite relations (see [16]). In logic programming languages [66], where the fundamental notion is that of predicate, we can immediately represent relations as such predicates. Since logic languages provide as primitives operations and features corresponding to predicate logic, it is quite straightforward to define relational calculus operations in terms of logic operations. The shape of the definitions in a logic language follows the form of Horn clauses, so we can just rewrite the naive set theory definitions of relational operations to adjust to this form.

3.3 Relations in databases

Relations are much used in database theory and standard practical implementations. In this context, relations have arbitrary finite arities (not just the binary ones found before), and their elements are usually considered to be unordered instead of tuples. Labels are associated to the sets involved as part of a relation, so there is a kind of typing environment involved in such database relations (the set of labels for a relation is called its scheme, and can intuitively be seen as its type). Leaving aside the matter of attributes and unordered structure for the time being, a database relation R is like a subset of a finite product $R \subseteq \Pi_{i \in I} A_i$ for some family of sets A_i.

This means that we are again in a situation where we need to handle subsets in type theory, so the discussion of the first section in this chapter still applies. The use of attributes can certainly complicate things, but it amounts in practice to an indirection level when getting to the types of the sets involved. How to represent the unordered structure of the tuple entries is a much harder problem, as will be seen in the chapter on databases in type theory later on.

Another concept of databases, much used in practical applications, is that of a key. This is a part of a relation's scheme that serves to identify the whole tuple in a unique way (for instance, a phone area prefix and a phone number for a relation storing the entries of a phone book). When a relation has a key, it can be seen as a kind of dictionary, as in the discussion in the previous section. The difference is that now the pairs (k, e) in the dictionary will be such that k is a value for the relation's key and e is the unique value for the rest of the single tuple associated with that key in the relation. Dictionary implementations can then be used for database relations that have a key. In practice, however, databases have a much more complex set of operations to implement than a simple dictionary does. Relational databases are implemented using some advanced data structures, for instance dynamic hash tables or B^+ trees.

Defining implementations of database relations in type theory and verifying their correctness with regards to the properties database operations have according to the relational model is a complex and interesting problem. The details and difficulty of such an effort will vary greatly according to the particular data structure we consider as an implementation. For instance, using a list-based representation (as we will show in a later chapter on relational database formalizations) is a non-trivial and interesting exercise, but not too hard in the context of this thesis. Doing the same for a realistic balanced tree-based representation, on the other hand, could be quite daunting.

3.4 Relational programming

There have been a few investigations of what a programming language paradigm based on the calculus of relations would be like and also how it would be implemented (see [16]). However, by relational programming we mean instead a programming logic based on the relation calculus and relational methods, and also relational semantics of programming languages.

As we will show in the following chapters, properly formalizing a version of the abstract calculus of relations in type theory will require quite some work. The several abstract notions will have to be adapted to type theory, particularly with respect to dealing with equality on the base sets of the abstract relations. In such a setting, the use of constructive category theory will be a great help, as it is a subject that has been explored in the existing literature (see for instance [41, 27]) . Once the abstract relational setting has been set up, the construction of its intended models (that is, concrete versions of mathematical relations) will also show the need for some adaptations, particularly when it comes to constructions reflecting power sets and their membership relations (much used in applying the abstract calculus of relations to program specification and derivation).

3.5 Representations for relations in type theory

We show here some Alfa definitions for several notions of relation inside type theory.

A relation can be a binary (curried) predicate:

$$PropRel\ (A, B\ \in\ Set) \in Type$$

$$PropRel\ A\ B \equiv A\ \rightarrow\ B\ \rightarrow\ Prop$$

Such a relation does not need to be decidable. However, $R\ \in\ A\ \rightarrow\ B\ \rightarrow\ Prop$ is decidable if we can prove $\forall A\ \in\ A.\, \forall B\ \in\ B.\, (R\, a\, b \vee \neg R\, a\, b)$. More generally, if we define

$$Decidable\ (P \in Prop) \in Prop$$

$$Decidable\ P \equiv P \vee (\neg P)$$

then we can define a decidable relation as:

$$DecPropRel\ (A, B\ \in\ Set) \in Type$$

$$DecPropRel\ A\ B \equiv \mathbf{sig} \left\{ \begin{array}{l} rel \in PropRel\ A\ B \\ dec \in (a\ \in\ A, b\ \in\ B) \rightarrow Decidable\ (rel\ a\ b) \end{array} \right\}$$

Alternatively, a decidable relation can be defined as a Boolean valued function:

$$BoolRel\ (A, B\ \in\ Set) \in Set$$

$$BoolRel\ A\ B \equiv A\ \rightarrow\ B\ \rightarrow\ Bool$$

This definition is equivalent to the previous one, except that now we only have a Boolean value indicating if $R\,a\,b$ holds or not, while in the previous definition we would have a proof object for either $R\,a\,b$ or for $\neg R\,a\,b$. The presence of such a proof object can make proofs of properties about a decidable relation much simpler than if we just had a Boolean value.

Often, we will be interested in finite relations. If we have a finite enumeration of the pairs in the relation, that is a function taking each element in a finite set to a pair, we get one representation of a finite relation:

$FinRel\ (A, B\ \in\ Set) \in Set$

$$FinRel\,A\,B \equiv \mathbf{sig}\ \left\{ \begin{array}{l} ar \in Nat \\ rel \in Fin\,ar\ \rightarrow\ Times\,A\,B \end{array} \right\}$$

Here *Times* is the non-dependent pair set constructor, provided by the Alfa library.

Alternatively, a finite relation can be represented by the list of its pairs:

$ListRel\ (A, B\ \in\ Set) \in Set$

$ListRel\,A\,B \equiv List\ (Times\,A\,B)$

An infinite countable relation can be represented by an enumeration of its pairs, that is a function taking each natural number to a pair in the relation:

$InfListRel\ (A, B\ \in\ Set) \in Set$

$InfListRel\,A\,B \equiv Nat\ \rightarrow\ Times\,A\,B$

We can also use the idea of adjacency lists, and represent a finite relation as a list of pairs of an element and all the elements that are related to it:

$AdjListRel\ (A, B\ \in\ Set) \in Set$

$AdjListRel\,A\,B \equiv List\ (Times\,A\ (List\,B))$

And also use the idea of adjacency matrix, which can be represented using the Alfa library notion of vector: $Vec\ (X \in Set)\ (n \in Nat) \in Set$ is the data type in the Alfa library providing vectors of elements of type X and length n. In this way, a finite decidable relation is a matrix is a vector of vector of Boolean values:

$AdjMatrRel\ (m, n\ \in\ Nat) \in Set$

$AdjMatrRel\,m\,n \equiv Vec\ (Vec\,Bool\,n)\ m$

3.6 On some difficulties which arise when expressing relations in Agda/Alfa

As shown by the preceding section, many different notions of concrete relation and simple data structure representations of relations can be expressed in a natural way in Alfa/Agda. Proving things about these different notions of relation can be done following the usual way of proving properties in this proof assistant, manipulating explicit proof objects following the Curry-Howard isomorphism.

Abstract relations, as used in relational algebras and categories, can be defined in this proof assistant by using signatures (dependent records). The definitions look clear and readable, but can seem a bit clumsy due to the absence of record subtyping in Alfa/Agda. Thus, to express that a signature A' represents some abstract notion that is a subclass of another defined by signature A we need to include a field in A' that is of type A. This can make the access to "inherited" fields from A a bit clumsy if we are not careful and use enough **open** sentences. **open** takes as argument a signature object and causes its fields to become accessible without the need of using dot notation.

As an example of how we need to do things in Alfa, consider the following: if we wish to define both the notions of preorder and partial order, we would first set up a **sig** *pr* for preorder:

$$pr \equiv \{A \in Set; R \in PropRel\ A\ A;\ isRefl \in \ldots;\ isTrans \in \ldots\}$$

Now we would need to set up a second **sig** *po* for partial order. As we can't make *po* a subtype of *pr*, the way to ensure that a value of type *po* is also a value of type *pr* is by putting a field in the definition of *po* having as type *pr*:

$$po \equiv \{PR \in pr;\ isAntisym \in \ldots\}$$

Access to the reflexivity proof for a value p of type *po* needs to be done by dot notation: $p.PR.isRefl$. When such nested **sig**s have several levels, as is usually the case when defining several related algebraic structures, the repeated dot notation can make things hard to read. A judicious use of **open** sentences can help, but such sentences can get in the way of the actual thing of interest that we are defining or handling.

An additional problem is that such algebras and categories require a lot of equational reasoning to prove properties about them. At present there is a lack of support for this in Alfa/Agda, resulting in lengthy proof objects that are hard to read and write even with the help of the layout and presentation features of the proof assistant.

For relations in databases, the new issue of how to represent attributes and their associated typing can be problematic. The general notion as used in relational database theory requires a global universe of attributes and their typings, with a relation scheme being an unordered set of attributes. Representing such unordered sets in Alfa/Agda is not trivial and can get quite difficult to properly capture once the associated typing is also taken into account. Simplifications allow the problem to become more manageable (for instance, making the typings local, or the schemes ordered lists of attributes). The best way of capturing the full notion of relations with schemes over attributes seems to be via an abstract definition of scheme, or even more usefully, finite sets with their properties. And once again, this abstraction needs a lot of equational reasoning if we wish to prove things about it. However, seen as just an interface to an abstract data type, the work becomes more reasonable as we just need to provide an implementation and prove that this implementation satisfies the axioms. Of course, for realistic data structures implementing the finite sets or schemes, the proofs will be very demanding and complex.

Within the particular subject this thesis investigates, type theory provides us with a language in which we can express both "mathematical" notions of relations and also "programming" notions of relations. These notions appear as part of a general framework for constructive mathematics where they interact in a natural way, both in a general sense (building on methods and ideas for doing constructive mathematics in type theory) and in a specific sense (building on the formalization of logic and general data types provided by the Alfa library and Alfa's particular features). This is then, in essence, the benefit of our work.

At this point, a valid question is what are the fundamental difficulties we have to address during the present work. One source of difficulty is that "sets" in type theory are not as flexible as "sets" in set theory. The sets we work with are more like data types in programming languages, and the practical issues that arise while working with them are also similar. For example, we mentioned the fact that we do not have subtyping. Another difficulty is that in type theory, the uniform equality set $Eq\ A\ a\ b$ (showing that $a = b$ for $a, b \in A$) is too "intensional" for many purposes and therefore we sometimes need to work with setoids (sets with equivalence relations) instead of sets. When one works with setoids, one needs to prove explicitly that functions preserve the equivalence relations on the setoids, and this creates some extra work. In our work, we use setoids for allegories and also for data base relations.

Finally, a possible objection from the point of view of a practical implementor of a database management system is that by using type theory we make our implementations in terms of a functional language, and this may lead to performance that is not good enough for the time-critical and detailed concurrent execution scheduling of the database management system. What's more, type theory (and the particular kind of type theory underlying Alfa/Agda) is not a standard functional language because of the need to stick to special recursion schemata which ensure termination of all functions, which could make such an objection even more reasonable. On the other hand, we can point to the current efficient implementations of functional languages available to the programmer, and also to the fact that in our work we don't concern ourselves with problems of timing, concurrency administration, or even the detailed low-level file I/O that is part of a database manager implementation.

Our concern is capturing the functionality of database operations when acting on some reasonable data structure implementing the relations, and verifying that this implementation (written in the functional language of type theory) satisfies the abstract specification of how the database operations should behave.

Yet another point to raise in regards to this objection is that it is also possible to reason about imperative languages in Agda/Alfa (that is, in Martin-Löf's type theory). One can for example implement types which represent the syntax and operational semantics. There is indeed quite a lot of work in this direction using the Coq system for reasoning about Javacard programs (see for instance [6]).

3.7 A guide to the rest of this chapter

We will now present in the following sections complete formalizations of relations and their operations in Alfa, and also of the standard properties a relation can satisfy.

First we deal with relations represented as propositional functions, that is, as entities of some type $A \to B \to Prop$. This means that to state $R\,a\,b$ holds is to say we have a proof object witnessing this fact. Definitions proceed by setting up and manipulating such proof objects according to the Curry-Howard isomorphism.

Then we deal with decidable relations in the sense that they are a binary boolean-valued function. Definitions proceed now by composing boolean operations that represent the usual meaning of relations and their operations.

Finally, we deal with propositional relations and boolean relations defined over sets that have an equality on their elements. This gives rise to the notion of E-relations, relations that respect the element equalities in a natural way.

3.8 Propositional Relations

3.8.1 Heterogeneous relations

A *heterogeneous relation* between sets A and B is a propositional function taking one argument from A followed by one argument from B.

$HetRel\ (A, B \in Set) \in Type$

$HetRel\,A\,B \equiv A \to B \to Prop$

We discussed the reason for this definition in 3.1. Let us just recall that by the Curry-Howard interpretation of propositions-as-types, this means that R holds on $a \in A$ and $b \in B$ exactly when $R\,a\,b$ is inhabited as a type. For short, many times we will just say relation when we mean heterogeneous relation, if there is no risk of confusion.

Regarding notation, please note that before we called this *PropRel*. That was done for the purposes of discussing representations of relations and give each style a descriptive name. Here, instead, we have chosen one representation for our work, and so decide to just call *HetRel* to the propositional relations.

Notation

As is usual practice, we will sometimes write $R\,a\,b$ as $a\,R\,b$.

3.8.2 Empty relation

Explanation

The *empty relation*, or *zero relation*, between sets A and B is the always false propositional function between them.

Definition

$\emptyset \in \mathit{HetRel}\,A\,B \qquad (A, B \in \mathit{Set})$

$\emptyset \equiv \lambda\, a\, b \;\rightarrow\; \bot$

3.8.3 Universal relation

Explanation

The *universal relation* between sets A and B is the always true propositional function over them.

Definition

$V \in \mathit{HetRel}\,A\,B \qquad (A, B \in \mathit{Set})$

$V \equiv \lambda\, a\, b \;\rightarrow\; \top$

3.8.4 Union of relations

Explanation

The *union* of two relations between sets A and B is the disjunction of their propositional functions.

Definition

$\cup\,(R, S \in \mathit{HetRel}\,A\,B) \in \mathit{HetRel}\,A\,B \qquad (A, B \in \mathit{Set})$

$R \cup S \equiv \lambda\, a\, b \rightarrow (a\, R\, b) \vee (a\, S\, b)$

3.8.5 Intersection of relations

Explanation

The *intersection* of two relations between sets A and B is the conjunction of their propositional functions.

Definition

$\cap\,(R, S \in \mathit{HetRel}\,A\,B) \in \mathit{HetRel}\,A\,B \qquad (A, B \in \mathit{Set})$

$R \cap S \equiv \lambda\, a\, b \rightarrow (a\, R\, b)\ \&\ (a\, S\, b)$

3.8.6 Complement of a relation

Explanation

The *complement* or *negation* of a relation is the negation of its propositional function.

Definition

$\neg\,(R \in \mathit{HetRel}\,A\,B) \in \mathit{HetRel}\,A\,B \qquad (A, B \in \mathit{Set})$

$\neg\, R \equiv \lambda\, a\, b \rightarrow \neg\,(a\, R\, b)$

Note the overloading of the symbol $\neg$, also used for the negation of propositions.

Discussion

It is important to remark that in our setting this complement will fail to satisfy many of the expected laws, because intuitionistic negation fails in the same way. For example, it is no longer true that $\neg\neg R = R$, because $\neg\neg R\, a\, b$ does not imply $R\, a\, b$.

3.8.7 Composition of relations

Explanation

The *composition* of a relation R between sets A and B and a relation S between sets B and C will be the relation between A and C that holds for $a \in A$ and $c \in C$ when there exists a bridging element $b \in B$ connecting them through R and S.

Definition

$\bullet\ (R \in \mathit{HetRel}\, A\, B, S \in \mathit{HetRel}\, B\, C) \in \mathit{HetRel}\, A\, C \qquad (A, B, C \in \mathit{Set})$

$R \bullet S \equiv \lambda\, a\, c \rightarrow \exists\, b \in B\,.\ (a\, R\, b)\ \&\ (b\, S\, c)$

3.8.8 Converse of a relation

Explanation

The *converse* of a relation between sets A and B is the same relation viewed the other way around, that is, as a relation between sets B and A.

Definition

$\circ\ (R \in \mathit{HetRel}\, A\, B) \in \mathit{HetRel}\, B\, A \qquad (A, B \in \mathit{Set})$

$R^{\circ} \equiv \lambda\, b\, a \rightarrow a\, R\, b$

3.8.9 Inclusion of relations

Explanation

A relation is *included* in another one if whenever it holds for two elements, the second relation also holds for those same elements.

Definition

$\subseteq (R, S \in \mathit{HetRel}\,A\,B) \in \mathit{Prop} \qquad (A, B \in \mathit{Set})$

$R \subseteq S \equiv \forall\, a \in A\,.\ \forall\, b \in B\,.\ (a\,R\,b) \supset (a\,S\,b)$

3.8.10 Equality of relations

Explanation

Two relations are *equal* if each one is included in the other one.

Definition

$= (R, S \in \mathit{HetRel}\,A\,B) \in \mathit{Prop} \qquad (A, B \in \mathit{Set})$

$R = S \equiv (R \subseteq S)\ \&\ (S \subseteq R)$

3.8.11 Left residual of relations

Explanation

Consider an inequation on relations, $X \bullet S \subseteq R$, where R, S are known and X is unknown. It would be nice to have an operation expressing one particular solution to this problem. The *left residual* of two relations provides the largest relation which is a solution to this inequation.

Definition

$/ (R \in \mathit{HetRel}\,A\,C, S \in \mathit{HetRel}\,B\,C) \in \mathit{HetRel}\,A\,B \qquad (A, B, C \in \mathit{Set})$

$R/S \equiv \lambda\, a\, b \rightarrow \forall\, c \in C\,.\ (b\,S\,c) \supset (a\,R\,c)$

3.8.12 Right residual of relations

Explanation

Consider an inequation on relations, $R \bullet X \subseteq S$, where R, S are known and X is unknown. In a similar way to the left residual, it would be nice to have an operation expressing one particular solution to this problem. The *right residual* of two relations provides the largest relation which is a solution to this inequation.

Definition

$\backslash\ (R \in \mathit{HetRel}\, A\, B, S \in \mathit{HetRel}\, A\, C) \in \mathit{HetRel}\, B\, C \qquad (A, B, C \in \mathit{Set})$

$R \backslash S \equiv \lambda\, b\, c \to \forall a \in A\, .\ (a\, R\, b) \supset (a\, S\, c)$

3.8.13 Homogeneous relations

Explanation

An *homogeneous relation* is a relation taking both arguments in the same set.

Definition

$\mathit{HomRel}\ A \in \mathit{Set} \in \mathit{Type}$

$\mathit{HomRel}\ A \equiv \mathit{HetRel}\ A\ A$

Since the library provides the same concept but implements it in a direct way (as a propositional function), we actually define *HomRel* as equivalent to the library's *Rel*. This doesn't cause us any problems as both definitions behave in the same way in Alfa.

3.8.14 Operations on homogeneous relations

All the operations discussed previously can be applied to homogeneous relations. While we could define new names by instantiating the operations, for instance, in this way:

$\cup'\ (R, S \in \mathit{HomRel}\, A) \in \mathit{HomRel}\, A \qquad (A \in \mathit{Set})$

$R \cup' S \equiv R \cup S$

this would only confuse things with excess notation. Given that Alfa allows us to hide the A over which the relation is defined, as indicated by the definition of *HetRel*, the extra definitions would not serve any aesthetic purpose either.

3.8.15 Properties of homogeneous relations

Relations can satisfy many interesting properties. The ones we define here are fundamental for all our work, but certainly don't make an exhaustive list. All the definitions are part of the existing library.

Reflexivity

$Reflexive\ (A \in Set, R \in Rel\ A) \in Prop$

$Reflexive\ A\ R \equiv (x \in A) \to x\ R\ x$

Symmetry

$Symmetrical\ (A \in Set, R \in Rel\ A) \in Prop$

$Symmetrical\ A\ R \equiv (x1, x2 \in A) \to x1\ R\ x2 \to x2\ R\ x1$

Transitivity

$Transitive\ (A \in Set, R \in Rel\ A) \in Prop$

$Transitive\ A\ R \equiv (x1, x2, x3 \in A) \to x1\ R\ x2 \to x2\ R\ x3 \to x1\ R\ x3$

Substitutivity

$Substitutive\ (A \in Set, R \in Rel\ A) \in Type$

$Substitutive\ A\ R \equiv (P \in Pred\ A, x1, x2 \in A) \to x1\ R\ x2 \to P\ x1 \to P\ x2$

3.8.16 Equivalence relations

A homogeneous relation is an *equivalence relation* if it is reflexive, symmetric and transitive. While we could define a signature type containing the required proof objects, for our purposes (and also following the Alfa library) we will skip the wrapping signature and use three proof objects corresponding to the properties.

3.8.17 Setoids

In constructive type theory quotienting is not a primitive set forming operation. If A is a set and $=_A$: $A \to A \to Set$ is an equivalence relation we cannot simply say $A/=_A = \{[x]_A \mid x \in A\}$ is a set.

Instead we introduce the notion of *setoid*, i.e., a type consisting of a set and an equivalence relation over that set. The underlying set is called the *carrier* of the setoid. If A is a setoid with equivalence relation $=_A$ and $='_A$ is an equivalence relation which is coarser than $=_A$ (i.e., $=_A \subseteq ='_A$), then we can define the setoid $A/='_A$ by replacing $=_A$ by $='_A$ as the equivalence relation on the setoid. Another name for setoid is *E-set*. In type theory setoids are one of the standard ways of dealing with equalities. The definition is part of the existing library.

$Setoid \in Type$

$$Setoid \equiv \begin{array}{l} \textbf{sig} \\ \quad Elem \in Set \\ \quad Equal \in Rel\; Elem \\ \quad ref \in (x \in Elem) \rightarrow Equal\; x\; x \\ \quad sym \in (x1, x2 \in Elem) \rightarrow Equal\; x1\; x2 \rightarrow Equal\; x2\; x1 \\ \quad tran \in (x1, x2, x3 \in Elem) \rightarrow \\ \qquad\qquad Equal\; x1\; x2 \rightarrow Equal\; x2\; x3 \rightarrow Equal\; x1\; x3 \end{array}$$

For a setoid X, its carrier *Elem* X is from now on denoted $|X|$, and its equivalence relation *Equal* X will be denoted $==_X$ (and used either in prefix or infix form depending on the context of use). This is in keeping with the conventions of the Alfa library.

3.8.18 Identity relation

Explanation

The *identity relation* over a setoid A is the relation that holds exactly for those pairs of elements in the carrier of A that are equal under the equivalence of A.

Definition

$I \in HomRel\; (|\,A\,|) \qquad (A \in Setoid)$

$I \equiv \lambda\, a1\; a2 \rightarrow a1 ==_A a2$

3.9 Boolean relations

3.9.1 Heterogeneous Boolean relations

A *heterogeneous Boolean relation* between sets A and B is a function taking an argument from A and an argument from B, and returning a Boolean value.

$BoolHetRel\; (A, B \in Set) \in Set$

$BoolHetRel\; A\; B \equiv A \rightarrow B \rightarrow Bool$

$R\,a\,b$ will always have as value one of the two Boolean values. This means that such a relation is decidable. Such relations can be programmed directly in a functional programming language, in contrast to the general relations of the previous chapter. Also note in the following the overloading of notation for constant Boolean relations and relational Boolean operations, as we use the same symbols as in the propositional relations case.

3.9.2 Constant Boolean relations

As in the previous chapter, we can define two special relations for the case where no elements are related, and the case where all elements are related. Now we need to use the Boolean values, however, instead of inhabited and non-inhabited sets.

Empty relation

$\emptyset \in \mathit{BoolHetRel}\,A\,B \qquad (A, B \in \mathit{Set})$

$\emptyset \equiv \lambda\,a\,b \rightarrow \textbf{false}$

Universal relation

$V \in \mathit{BoolHetRel}\,A\,B \qquad (A, B \in \mathit{Set})$

$V \equiv \lambda\,a\,b \rightarrow \textbf{true}$

3.9.3 Operations on Boolean relations

We can define operations on Boolean relations that are analogous to the operations on general relations of the previous chapter. But in this case we use Boolean operators instead of the logical constants.

Union

$\cup\,(R, S \in \mathit{BoolHetRel}\,A\,B) \in \mathit{BoolHetRel}\,A\,B \qquad (A, B \in \mathit{Set})$

$R \cup S \equiv \lambda\,a\,b \rightarrow R\,a\,b \vee S\,a\,b$

Intersection

$\cap\,(R, S \in \mathit{BoolHetRel}\,A\,B) \in \mathit{BoolHetRel}\,A\,B \qquad (A, B \in \mathit{Set})$

$R \cap S \equiv \lambda\,a\,b \rightarrow R\,a\,b \wedge S\,a\,b$

Complement

$\neg\,(R \in \mathit{BoolHetRel}\,A\,B) \in \mathit{BoolHetRel}\,A\,B \qquad (A, B \in \mathit{Set})$

$\neg\,R \equiv \lambda\,a\,b \rightarrow \neg\,(R\,a\,b)$

Converse

$\circ\,(R \in \mathit{BoolHetRel}\,A\,B) \in \mathit{BoolHetRel}\,B\,A \qquad (A, B \in \mathit{Set})$

$R^{\circ} \equiv \lambda\,b\,a \rightarrow R\,a\,b$

Composition

Boolean relations are not composable in general. The bridging element can belong to an arbitrary infinite set, and hence the existential quantification involving it would not be decidable in general. To get around this problem, we need to limit ourselves to Boolean relations over finite sets.

3.9.4 Finite Boolean relations

A *finite Boolean relation* is a Boolean relation between two finite sets *Fin m* and *Fin n*.

3.9.5 Composition of finite Boolean relations

Definition

$\bullet\,(R \in \mathit{BoolHetRel}\,(\mathit{Fin}\,m)\,(\mathit{Fin}\,n), S \in \mathit{BoolHetRel}\,(\mathit{Fin}\,n)\,(\mathit{Fin}\,p)) \in$
$\quad \mathit{BoolHetRel}\,(\mathit{Fin}\,m)\,(\mathit{Fin}\,p) \qquad (m, n, p \in \mathit{Nat})$

$R \bullet S \equiv \lambda\,a\,c \rightarrow \exists b \in \mathit{Fin}\,n\,.\,R\,a\,b \wedge S\,b\,c$

Discussion

It is important to remark that the existential quantifier here is a decidable one, with domain a finite set of the form *Fin n* and result a Boolean value. The Alfa library provides us with it.

3.9.6 Comparing Boolean relations

As in the previous chapter, we can define inclusion and equality of Boolean relations. These will still be propositions, so it is required to lift the Boolean values of relation instances to *Prop*. However, when comparing finite Boolean relations we have computable inclusion and equality.

Inclusion

$\subseteq (R, S \in \mathit{BoolHetRel}\, A\, B) \in \mathit{Prop} \qquad (A, B \in \mathit{Set})$

$R \subseteq S \equiv \forall a \in A\,.\, \forall b \in B\,.\, (|R\, a\, b|) \supset (|S\, a\, b|)$

Equality

$= (R, S \in \mathit{BoolHetRel}\, A\, B) \in \mathit{Prop} \qquad (A, B \in \mathit{Set})$

$R = S \equiv (R \subseteq S)\ \&\ (S \subseteq R)$

3.9.7 Homogeneous Boolean relations

A *homogeneous Boolean relation* is a Boolean relation taking both arguments in the same set.

Definition

$\mathit{BoolHomRel}\, (A \in \mathit{Set}) \in \mathit{Set}$

$\mathit{BoolHomRel}\, A \equiv \mathit{BoolHetRel}\, A\, A$

The existing Alfa library already provides such homogeneous Boolean relations, calling them *rel*.

3.9.8 Lifted Boolean relations

In many situations, we need to assert that a Boolean relation holds. This is done by requiring a proof object for the lifted Boolean relation $|R\, a\, b|$.

3.9.9 Datoids

Explanation

A *datoid* is a set provided with a Boolean equality relation that is reflexive and substitutive. In this case, reflexivity and substitutivity refer to the lifted relation instead of using new and specific Boolean versions of these properties. The concept of datoid is a standard way in the Alfa library of dealing with sets with decidable substitutive equalities.

Definition

$Datoid \in Type$

$$Datoid \equiv \mathbf{sig} \left\{ \begin{array}{l} Elem \in Set \\ eq \in rel\, Elem \\ ref \in (x \in Elem) \rightarrow |\, eq\, x\, x\, | \\ subst \in Substitutive\, Elem\, (\lambda\, x1\, x2 \rightarrow |\, eq\, x1\, x2\, |) \end{array} \right\}$$

3.9.10 Identity Boolean relation

As for setoids, the *identity Boolean relation* for a datoid coincides with its Boolean equality. We will not have need of a special definition, however.

3.10 E-relations

3.10.1 Introduction

If we wish to define a notion of relation R between two setoids A and B, it is natural to ask that the equivalence relations $==_A$ and $==_B$ are respected. We therefore introduce the notion of E-relation.

3.10.2 E-relations

Explanation

An *E-relation* between two setoids A and B is a relation between their carrier sets (that is, a propositional binary function) such that substituting equal elements in an instance for which the relation holds will give an instance that also holds.

Definition

$$ERelation \begin{pmatrix} A \in Setoid \\ B \in Setoid \end{pmatrix} \in Type$$

$$ERelation\ A\ B \equiv \quad \begin{array}{l} \textbf{sig} \\ \quad rel \in HetRel\ (|\,A\,|)\ (|\,B\,|) \\ \quad resp \in \forall\, a1\ \in |A|.\ \forall\, a2\ \in |A|.\ \forall\, b1\ \in |B|.\ \forall\, b2\ \in |B|. \\ \qquad ==_A\ a1\, a2\ \rightarrow ==_B\ b1\, b2\ \rightarrow\ rel\, a1\, b1\ \rightarrow\ rel\, a2\, b2 \end{array}$$

Discussion

We should remark that the use of expressions such as $==_A\ a1\, a2$, while perhaps unusual-looking to the reader, is just the usual prefix relation application to two arguments. Here it just happens that the relation is an equality coming from a setoid. In Haskell, $(==_A)$ is used for the prefix notation of an infix symbol. Here, we are actually using Alfa's feature of displaying names as symbols instead.

In what follows, we refer to the definitions for propositional relation constants and operators. We remark also that, in the rest of this work, a "proof" heading indicates that the following text is an Alfa/Agda proof object. Such proof objects have the form of an Alfa definition, with a first line showing the name, arguments, and type of the object, and the rest of the text showing the actual proof structure. Finally, as a useful notational convention, we will refer to the E-relational constant or operator corresponding to a relational one by a trailing equality sign after the basic relational name presented before in this chapter.

3.10.3 Constants for E-relations

Empty E-relation

Proposition The *empty relation* between two setoids is an E-relation.

Proof

$$\emptyset = \ \in ERelation\ A\ B \qquad \begin{pmatrix} A \in Setoid \\ B \in Setoid \end{pmatrix}$$

$$\emptyset = \equiv \quad \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} rel \equiv \emptyset \\ resp \equiv \lambda\, a1\, a2\, b1\, b2\, hA\, hB\, h\ \rightarrow\ elimAbsurd\ (rel\, a2\, b2)\ h \end{array} \right. \end{array}$$

The above proof-object resorts to a use of the rule for absurd elimination in the standard Alfa library. The first argument is the desired proposition to be proved, and the second argument an absurd proposition.

Universal E-relation

Proposition The *universal relation* between two setoids is an E-relation.

Proof

$$V\!= \;\in ERelation\,A\,B \qquad \left(\begin{array}{l} A \in Setoid \\ B \in Setoid \end{array}\right)$$

$$V\!= \equiv \quad \mathbf{struct}\left[\begin{array}{l} rel \equiv V \\ resp \equiv \lambda\, a1\, a2\, b1\, b2\, hA\, hB\, h \;\rightarrow\; \mathbf{tt} \end{array}\right.$$

In the above proof object, the proof that the universal relation respects equality is trivial, as shown by the **tt**.

Identity E-relation

Proposition The *identity* relation on a setoid is an E-relation on that setoid.

Proof

$$I\!= \;\in ERelation\,A\,A \qquad (A \;\in\; Setoid)$$

$$I\!= \equiv \quad \mathbf{struct}\left[\begin{array}{l} rel \equiv I \\ resp \equiv \lambda\, a1\, a2\, a3\, a4\, h\, h'\, h'' \;\rightarrow \\ \qquad\qquad A.tran\, a2\, a3\, a4\; (A.tran\, a2\, a1\, a3\; (A.sym\, a1\, a2\, h)\; h'')\; h' \end{array}\right.$$

The proof builds a transitive chain of equalities from $a1 == a2$, $a3 == a4$ and $a1 == a3$, to prove that $a2 == a4$. This is the typical structure of a proof showing that a particular relational operator satisfies the definition of E-relation, and the reader should keep this in mind as a hint to make easier the reading of all similar proofs in this section.

3.10.4 Operations on E-relations

For every relational operator, its corresponding E-relation operator is defined as the relational one acting on the underlying relations, together with a proof that we can substitute equal elements for equal elements in the element argument places.

Union

Proposition The *union* of two E-relations between two setoids is also an E-relation between the same setoids.

Proof

$$\cup = \left(\begin{array}{l} R \in ERelation\ A\ B \\ S \in ERelation\ A\ B \end{array} \right) \in ERelation\ A\ B \qquad \left(\begin{array}{l} A \in Setoid \\ B \in Setoid \end{array} \right)$$

$$R \cup = S \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} rel \equiv R.rel\ \cup\ S.rel \\ resp \equiv \lambda\, a1\, a2\, b1\, b2\, hA\, hB\, h\ \rightarrow \\ \qquad\qquad \textbf{case}\ h\ \textbf{of} \\ \qquad\qquad\quad \textbf{inl}\ byR\ \rightarrow \\ \qquad\qquad\qquad \textbf{inl}\ (R.resp\, a1\, a2\, b1\, b2\, hA\, hB\, byR) \\ \qquad\qquad\quad \textbf{inr}\ byS\ \rightarrow \\ \qquad\qquad\qquad \textbf{inr}\ (S.resp\, a1\, a2\, b1\, b2\, hA\, hB\, byS) \end{array} \right. \end{array}$$

The proof simply examines which of R and S is actually holding for $a1$ and $b1$, and then applies the property of R or S being an E-relation, correspondingly.

Intersection

Proposition The *intersection* of two E-relations between two setoids is also an E-relation between the same setoids.

Proof

$$\cap = \left(\begin{array}{l} R \in ERelation\ A\ B \\ S \in ERelation\ A\ B \end{array} \right) \in ERelation\ A\ B \qquad \left(\begin{array}{l} A \in Setoid \\ B \in Setoid \end{array} \right)$$

$$R \cap = S \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} rel \equiv R.rel\ \cap\ S.rel \\ resp \equiv \lambda\, a1\, a2\, b1\, b2\, hA\, hB\, h\ \rightarrow \\ \qquad\qquad \textbf{struct} \\ \qquad\qquad\quad \left[\begin{array}{l} fst \equiv R.resp\, a1\, a2\, b1\, b2\, hA\, hB\, h.fst \\ snd \equiv S.resp\, a1\, a2\, b1\, b2\, hA\, hB\, h.snd \end{array} \right. \end{array} \right. \end{array}$$

The proof consists of substituting $a2, b2$ in place of $a1, b1$ in the original conjunction that proves the relational intersection holds for $a1, b1$.

Complement

Proposition The complement of an E-relation between two setoids is also an E-relation between the same setoids.

Proof

$$\neg = (R \in ERelation\ A\ B) \in ERelation\ A\ B \qquad \begin{pmatrix} A \in Setoid \\ B \in Setoid \end{pmatrix}$$

$$\neg = R \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} rel \equiv \neg\, R.rel \\ resp \equiv \lambda\, a1\, a2\, b1\, b2\, hA\, hB\, h\, h' \to \\ \quad h\ (R.resp\, a2\, a1\, b2\, b1\ (A.sym\, a1\, a2\, hA)\ (B.sym\, b1\, b2\, hB)\ h') \end{array} \right. \end{array}$$

The proof constructs an object for $R\, a2\, b2 \to \bot$ by applying the existing proof of $R\, a1\, b1 \to \bot$ to the result of substituting $a1, b1$ for $a2, b2$ within the argument for this new object.

Composition

Proposition The *composition* of an E-relation between setoids A and B with an E-relation between setoids B and C is an E-relation between A and C.

Proof

$$\bullet = \begin{pmatrix} R \in ERelation\ A\ B \\ S \in ERelation\ B\ C \end{pmatrix} \in ERelation\ A\ C \qquad \begin{pmatrix} A \in Setoid \\ B \in Setoid \\ C \in Setoid \end{pmatrix}$$

$$R \bullet = S \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} rel \equiv R.rel\ \bullet\ S.rel \\ resp \equiv \lambda\, a1\, a2\, c1\, c2\, hA\, hC\, h \to \\ \qquad \textbf{struct} \\ \qquad \left[\begin{array}{l} fst \equiv h.fst \\ snd \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv R.resp\, a1\, a2\, h.fst\, h.fst\, hA \\ \qquad\qquad (B.ref\, h.fst)\ h.snd.fst \\ snd \equiv S.resp\, h.fst\, h.fst\, c1\, c2 \\ \qquad\qquad (B.ref\, h.fst)\ hC\, h.snd.snd \end{array} \right. \end{array} \end{array} \right. \end{array} \right. \end{array}$$

The proof builds the new object for showing that the composition holds for $a2, c2$ by using the E-relation property of both R and S, and picking as the bridging element the same one that made true the composition for $a1, c1$.

Converse

Proposition The *converse* of an E-relation between setoids A and B is an E-relation between setoids B and A.

Proof

$$\circ = (R \in ERelation\,A\,B) \in ERelation\,B\,A \qquad \begin{pmatrix} A \in Setoid \\ B \in Setoid \end{pmatrix}$$

$$R^{\circ} = \equiv \quad \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} rel \equiv R.rel^{\circ} \\ resp \equiv \lambda\, b1\, b2\, a1\, a2\, hB\, hA\, h \;\rightarrow\; R.resp\, a1\, a2\, b1\, b2\, hA\, hB\, h \end{array} \right. \end{array}$$

The proof is immediate, since swapping the order of the arguments of R doesn't alter its E-relation property.

Right residual

Proposition The *right residual* of an E-relation between setoids A and B with an E-relation between setoids A and C is an E-relation between setoids B and C.

Proof

$$\backslash = \begin{pmatrix} R \in ERelation\,A\,B \\ S \in ERelation\,A\,C \end{pmatrix} \in ERelation\,B\,C \qquad \begin{pmatrix} A \in Setoid \\ B \in Setoid \\ C \in Setoid \end{pmatrix}$$

$$R \backslash = S \equiv \quad \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} rel \equiv R.rel \backslash S.rel \\ resp \equiv \lambda\, b1\, b2\, c1\, c2\, hB\, hC\, h\, a\, h' \;\rightarrow \\ \qquad\qquad S.resp\, a\, a\, c1\, c2\; (A.ref\, a)\; hC \\ \qquad\qquad\quad (h\, a\; (R.resp\, a\, a\, b2\, b1\; (A.ref\, a)\; (B.sym\, b1\, b2\, hB)\; h')) \end{array} \right. \end{array}$$

The proof builds an object for the universally quantified implication that constitutes the definition of a right residual, with arguments a and $h' \in R\,a\,c2$. It then builds a proof of $S\,a\,c2$ using the facts that R and S are E-relations.

Left residual

Proposition The *left residual* of an E-relation between setoids A and C with an E-relation between setoids B and C is an E-relation between A and B.

Proof

$$/\!= \begin{pmatrix} R \in ERelation\ A\ C \\ S \in ERelation\ B\ C \end{pmatrix} \in ERelation\ A\ B \qquad \begin{pmatrix} A \in Setoid \\ B \in Setoid \\ C \in Setoid \end{pmatrix}$$

$$R /\!= S \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} rel \equiv R.rel / S.rel \\ resp \equiv \lambda\, a1\ a2\ b1\ b2\ hA\ hB\ h\ c\ h' \rightarrow \\ \qquad\qquad R.resp\ a1\ a2\ c\ c\ hA\ (C.ref\ c) \\ \qquad\qquad\quad (h\ c\ (S.resp\ b2\ b1\ c\ c\ (B.sym\ b1\ b2\ hB)\ (C.ref\ c)\ h')) \end{array}\right. \end{array}$$

The proof is very similar to the one for right residual, changing only in the particular implication now needed for the right residual.

3.10.5 Comparing E-relations

The comparisons for inclusion and equality of E-relations are simply lifted from their relation parts.

Inclusion

$$\subseteq \begin{pmatrix} R \in ERelation\ A\ B \\ S \in ERelation\ A\ B \end{pmatrix} \in Prop \qquad \begin{pmatrix} A \in Setoid \\ B \in Setoid \end{pmatrix}$$

$$R \subseteq S \equiv R.rel \subseteq S.rel$$

Equality

$$= \begin{pmatrix} R \in ERelation\ A\ B \\ S \in ERelation\ A\ B \end{pmatrix} \in Prop \qquad \begin{pmatrix} A \in Setoid \\ B \in Setoid \end{pmatrix}$$

$$R = S \equiv R.rel = S.rel$$

3.11 Boolean E-relations

3.11.1 Boolean E-relations

Explanation

A *Boolean E-relation* between two setoids A and B is a boolean relation between their carrier sets such that replacing equal elements in an instance for which the relation holds will give an instance that also holds. The assertion that the Boolean relation holds on an instance is by lifting the relation instance.

Definition

$$BoolERel \left(\begin{array}{l} A \in Setoid \\ B \in Setoid \end{array} \right) \in Type$$

$$BoolERel\ A\ B \equiv \begin{array}{l} \textbf{sig} \\ \quad rel \in BoolHetRel\ (|\,A\,|)\ (|\,B\,|) \\ \quad resp \in \forall\, a1\ \in |A|.\ \forall\, a2\ \in |A|.\ \forall\, b1\ \in |B|.\ \forall\, b2\ \in |B|. \\ \qquad ==_A\ a1\, a2\ \rightarrow ==_B\ b1\, b2\ \rightarrow\ (|\, rel\, a1\, b1\,|)\ \rightarrow\ |\, rel\, a2\, b2\,| \end{array}$$

Notation

We remark that in the rest of this section we will once more overload notation. The constants and operators for Boolean E-relations will have the same names that their corresponding propositional E-relations have.

3.11.2 Special Boolean E-relations

In this subsection and the ones that follow, we refer to the constants and operators for Boolean relations from earlier on in this chapter.

Empty Boolean E-relation

Proposition The *empty Boolean relation* between two setoids is a Boolean E-relation.

Proof

$$\emptyset = \ \in BoolERel\ A\ B \qquad \left(\begin{array}{l} A \in Setoid \\ B \in Setoid \end{array} \right)$$

$$\emptyset = \equiv \quad \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} rel \equiv \emptyset \\ resp \equiv \lambda\, a1\, a2\, b1\, b2\, hA\, hB\, h\ \rightarrow\ elimAbsurd\ (|\, rel\, a2\, b2\,|)\ h \end{array} \right. \end{array}$$

Universal Boolean E-relation

Proposition The *universal Boolean relation* between two setoids is a Boolean E-relation.

Proof

$$V\!\!= \;\in BoolERel\; A\; B \qquad \begin{pmatrix} A \in Setoid \\ B \in Setoid \end{pmatrix}$$

$$V\!\!= \equiv \quad \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} rel \equiv V \\ resp \equiv \lambda\, a1\, a2\, b1\, b2\, hA\, hB\, h \;\rightarrow\; \textbf{tt} \end{array} \right. \end{array}$$

Identity Boolean E-relation

Proposition The *identity Boolean relation* on a finite set is a Boolean E-relation.

Proof

$$I\;(n\;\in\;Nat) \in BoolERel\;\;(Fin\,n)\;\;(Fin\,n)$$

$$I\,n \equiv \quad \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} rel \equiv \lambda\, a\, a' \;\rightarrow\; eqFin\, n\, a\, a' \\ resp \equiv \lambda\, a1\, a2\, b1\, b2\, hA\, hB\, h \;\rightarrow \\ \qquad\qquad substEq \\ \qquad\qquad\quad n \\ \qquad\qquad\quad \lambda\, a \;\rightarrow\; |\, rel\, a\, b2\, | \\ \qquad\qquad\quad a1 \\ \qquad\qquad\quad a2 \\ \qquad\qquad\quad hA \\ \qquad\qquad\quad (substEq\, n\, \lambda\, b \;\rightarrow\; |\, rel\, a1\, b\, |\, b1\, b2\, hB\, h) \end{array} \right. \end{array}$$

This proof is based on the substitutivity of the equality for finite setoids, a fact proved as part of the standard Alfa library as *substEq*. Note that since substitutivity refers to predicates over a single argument, a nested use of this property is made inside another one, one use for each argument position of the equality relation.

3.11.3 Operations on Boolean E-relations

Union

Proposition The union of two Boolean E-relations between two setoids is also a Boolean E-relation between the same setoids.

Proof

$$\cup = \left(\begin{array}{l} R \in \mathit{BoolERel}\ A\ B \\ S \in \mathit{BoolERel}\ A\ B \end{array} \right) \in \mathit{BoolERel}\ A\ B \qquad \left(\begin{array}{l} A \in \mathit{Setoid} \\ B \in \mathit{Setoid} \end{array} \right)$$

$$R \cup = S \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} rel \equiv R.rel \cup S.rel \end{array} \right. \\ \left[\begin{array}{l} resp \equiv \lambda\, a1\, a2\, b1\, b2\, hA\, hB\, h \rightarrow \\ \quad \textbf{case}\ (spec\_or\ (R.rel\, a1\, b1)\ (S.rel\, a1\, b1)).fst\, h\ \textbf{of} \\ \quad\quad \textbf{inl}\, inR \rightarrow \begin{array}{l} intro\_or\_lft \\ \ (R.rel\, a2\, b2) \\ \ (S.rel\, a2\, b2) \\ \ (R.resp\, a1\, a2\, b1\, b2\, hA\, hB\, inR) \end{array} \\ \quad\quad \textbf{inr}\, inS \rightarrow \begin{array}{l} intro\_or\_rgt \\ \ (R.rel\, a2\, b2) \\ \ (S.rel\, a2\, b2) \\ \ (S.resp\, a1\, a2\, b1\, b2\, hA\, hB\, inS) \end{array} \end{array} \right. \end{array}$$

In this proof is important to remark the difference with the previous section. Since we deal with boolean values here, we don't have a structured proof object to look into. Instead, we use the Alfa library's introduction (*intro_or_lft* and *intro_or_rgt*) and transformation (*spec_or*) rules for boolean valued logic. The reader should keep this in mind as a hint for an easier reading of the following proofs in this section. In this case, *spec_or.fst* creates a value in a sum type corresponding to the boolean disjunction argument.

Intersection

Proposition The intersection of two Boolean E-relations over two setoids is also a Boolean E-relation between the same setoids.

Proof

$$\cap = \left(\begin{array}{l} R \in \mathit{BoolERel}\ A\ B \\ S \in \mathit{BoolERel}\ A\ B \end{array} \right) \in \mathit{BoolERel}\ A\ B \qquad \left(\begin{array}{l} A \in \mathit{Setoid} \\ B \in \mathit{Setoid} \end{array} \right)$$

$$\begin{array}{l} R \cap = S \equiv \\ \quad \textbf{struct} \\ \quad\quad \left[\begin{array}{l} rel \equiv R.rel \cap S.rel \end{array} \right. \\ \quad\quad \left[\begin{array}{l} resp \equiv \lambda\, a1\, a2\, b1\, b2\, hA\, hB\, h \rightarrow \\ \quad \textbf{let} \left[\begin{array}{l} aux \in (|\, R.rel\, a1\, b1\, |)\ \&\ (|\, S.rel\, a1\, b1\, |) \\ \ aux \equiv (spec\_and\ (R.rel\, a1\, b1)\ (S.rel\, a1\, b1)).fst\, h \end{array} \right. \\ \quad \textbf{in}\ intro\_and\ (R.rel\, a2\, b2)\ (S.rel\, a2\, b2) \\ \quad\quad (R.resp\, a1\, a2\, b1\, b2\, hA\, hB\, aux.fst) \\ \quad\quad (S.resp\, a1\, a2\, b1\, b2\, hA\, hB\, aux.snd) \end{array} \right. \end{array}$$

In this proof *spec_and.fst* transforms a boolean conjunction into the pair-shaped proof object for a propositional conjunction.

Converse

Proposition The converse of a Boolean E-relation is also a Boolean E-relation.

Proof

$$\circ = (R \in \mathit{BoolERel}\ A\ B) \in \mathit{BoolERel}\ B\ A \qquad \begin{pmatrix} A \in \mathit{Setoid} \\ B \in \mathit{Setoid} \end{pmatrix}$$

$$R^{\circ =} \equiv \quad \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} \mathit{rel} \equiv R.\mathit{rel}^{\circ} \\ \mathit{resp} \equiv \lambda\, a1\ a2\ b1\ b2\ hA\ hB\ h\ \rightarrow\ R.\mathit{resp}\ b1\ b2\ a1\ a2\ hB\ hA\ h \end{array} \right. \end{array}$$

Complement

Proposition The complement of a Boolean E-relation is also a Boolean E-relation.

Proof

$$\neg = (R \in \mathit{BoolERel}\ A\ B) \in \mathit{BoolERel}\ A\ B \qquad \begin{pmatrix} A \in \mathit{Setoid} \\ B \in \mathit{Setoid} \end{pmatrix}$$

$$\begin{array}{l} \neg = R \equiv \\ \quad \textbf{struct} \\ \qquad \left[\ \mathit{rel} \equiv \neg\, R.\mathit{rel} \right. \\ \qquad \left[\ \mathit{resp} \equiv \lambda\, a1\ a2\ b1\ b2\ hA\ hB\ h\ \rightarrow \begin{array}{c} (\mathit{spec\_not}\ (R.\mathit{rel}\ a2\ b2))\ .\mathit{snd} \\ \lambda\, h' \rightarrow \begin{pmatrix} \mathit{spec\_not} \\ (R.\mathit{rel}\ a1\ b1) \end{pmatrix} .\mathit{fst}\ h \\ \begin{pmatrix} R.\mathit{resp} \\ a2 \\ a1 \\ b2 \\ b1 \\ (A.\mathit{sym}\ a1\ a2\ hA) \\ (B.\mathit{sym}\ b1\ b2\ hB) \\ h' \end{pmatrix} \end{array} \right. \end{array}$$

This proof follows the same idea used in the proof for the E-relation complement of the previous section. The only difference is the use of the transformation *spec_not* to move back and forth between boolean complements and propositional ones.

Composition

Proposition The composition of two Boolean E-relations over finite sets is also a Boolean E-relation.

Proof

$$\bullet = \left(\begin{array}{l} R \in \mathit{BoolERel}\ (\mathit{Fin}\, m)\ (\mathit{Fin}\, n) \\ S \in \mathit{BoolERel}\ (\mathit{Fin}\, n)\ (\mathit{Fin}\, p) \end{array} \right) \in \mathit{BoolERel}\ (\mathit{Fin}\, m)\ (\mathit{Fin}\, p) \qquad \left(\begin{array}{l} m \in \mathit{Nat} \\ n \in \mathit{Nat} \\ p \in \mathit{Nat} \end{array} \right)$$

$$R \bullet = S \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} \mathit{rel} \equiv R.\mathit{rel}\ \bullet\ S.\mathit{rel} \\ \mathit{resp} \equiv \lambda\, a1\, a2\, c1\, c2\, hA\, hC\, h\ \rightarrow \\ \quad \mathit{substEq}\, m\, (\lambda\, a\ \rightarrow\ |\, \mathit{rel}\, a\, c2\, |)\, a1\, a2\, hA \\ \quad\quad (\mathit{substEq}\, p\, (\lambda\, c\ \rightarrow\ |\, \mathit{rel}\, a1\, c\, |)\, c1\, c2\, hC\, h) \end{array} \right. \end{array}$$

3.11.4 Comparing Boolean E-relations

The comparisons between Boolean E-relations over setoids are lifted directly from the comparisons between their Boolean relation parts.

Inclusion

$$\subseteq \left(\begin{array}{l} R \in \mathit{BoolERel}\, A\, B \\ S \in \mathit{BoolERel}\, A\, B \end{array} \right) \in \mathit{Prop} \qquad \left(\begin{array}{l} A \in \mathit{Setoid} \\ B \in \mathit{Setoid} \end{array} \right)$$

$$R\ \subseteq\ S \equiv R.\mathit{rel}\ \subseteq\ S.\mathit{rel}$$

Equality

$$= \left(\begin{array}{l} R \in \mathit{BoolERel}\, A\, B \\ S \in \mathit{BoolERel}\, A\, B \end{array} \right) \in \mathit{Prop} \qquad \left(\begin{array}{l} A \in \mathit{Setoid} \\ B \in \mathit{Setoid} \end{array} \right)$$

$$R = S \equiv R.\mathit{rel} = S.\mathit{rel}$$

3.11.5 Finite sets and Boolean E-relations

Proposition

Any Boolean relation between two finite sets is a Boolean E-relation between the setoids obtained from those sets.

Proof

$$\mathit{finRelToERel} \left(\begin{array}{l} m \in \mathit{Nat} \\ n \in \mathit{Nat} \\ R \in \mathit{BoolHetRel}\ (\mathit{Fin}\, m)\ (\mathit{Fin}\, n) \end{array} \right) \in$$
$$\mathit{BoolERel}\ (\mathit{FinSetoid.Fin}\, m)\ (\mathit{FinSetoid.Fin}\, n)$$

$$\begin{array}{l} \mathit{finRelToERel}\, m\, n\, R \equiv \\ \quad \textbf{struct} \\ \qquad \left[\mathit{rel} \equiv R \right. \\ \qquad \left[\begin{array}{l} \mathit{resp} \equiv \lambda\, a1\, a2\, b1\, b2\, hA\, hB\, h \ \rightarrow \\ \qquad \mathit{substEq}\, m \\ \qquad\quad (\lambda\, a \ \rightarrow\ |\, \mathit{rel}\, a\, b2\, |)\, a1\, a2\, hA\ \ (\mathit{substEq}\, n\, (\lambda\, b\ \rightarrow\ |\, \mathit{rel}\, a1\, b\, |)\, b1\, b2\, hB\, h) \end{array} \right. \end{array}$$

This proof is based in the substitutivity of the equality for finite setoids. This works by using a pair of nested calls to the substitutivity predicate, using as predicates partial instantiations of the relation (that is, one element is fixed, being one of the original four ones, and the other becomes the argument of the predicate we perform the substitution on).

Chapter 4

Categories and Allegories

4.1 Introduction

An allegory is a special kind of category where the objects and arrows are to be thought of as abstract sets and relations. Composition becomes abstract relational composition, and there are also new operations corresponding to abstract intersection and converse of relations, plus a preorder (abstract inclusion of relations) on arrows consistent with the equality of the category (and a few equations and inequations capturing the properties of these new operations and preorder). The theory of allegories (and its several extensions, where additional operators like abstract relation union appear) provides us with a calculus of relations.

How do we formalize the notion of an allegory in constructive type theory? Since it is a special kind of category we can rely on already existing ideas about how to formalize category theory in constructive type theory, see for instance [41, 59, 27, 9]. These efforts have converged on the definition of an E-category. This is a structure where we have a *set* of objects, and for each pair of objects we have a *setoid* of arrows. This structure provides us with an abstract notion of equality of arrows but not with an abstract notion of equality of objects. Composition of arrows must preserve equality of arrows, and the category laws are valid with respect to the notion of equality of arrows. The cited works show that this definition is suitable for developing interesting parts of elementary category theory in type theory.

So in what follows we will introduce a notion of E-allegory essentially by extending the notion of an E-category with some extra structure. While this is the idea and the goal, it will be formally simpler to take a slightly different route. Since the axioms of an allegory refer to an abstract notion of inclusion of arrows, we will start by introducing the notion of an LT-category where instead of a setoid of arrows we have a set with a preorder of arrows. The preorder is our abstract notion of inclusion. Equality in an LT-category will then be a derived notion: two arrows are equal iff they are included in each other.

Even more, to structure the definition of an LT-category we split it in two: first we introduce the notion of a *precategory*, which is a signature which includes the operations of an LT-category. Then we get the full notion of an LT-category by adding the axioms to it. Furthermore, we consider large LT-categories, rather than small. Just like in set theory categorical notions come in both large and small versions. But in constructive type theory we use the set/type distinction rather than the set/class distinction of set theory. Since the notion of large precategory has a component which is a type (i.e. a member of #1), it has to be a member of a larger kind #2. It is a "superlarge" notion.

4.2 LT-precategories

A *large LT-precategory* is like the algebraic signature of a category based on a preordering of arrows (instead of equality of arrows). As such, it provides a type of objects, a type of arrows over each pair of objects, identity arrows for each object, and the operation of arrow composition. It does not provide any laws regarding these types and operations, though.

The definition in Alfa is:

$$\mathit{LargePrecat} \in \#2$$

$$\mathit{LargePrecat} \equiv \begin{array}{l} \textbf{sig} \\ \quad \mathit{obj} \in \mathit{Type} \\ \quad \mathit{arr} \in \mathit{obj} \to \mathit{obj} \to \mathit{Type} \\ \quad \mathit{lt} \in (A, B \in \mathit{obj}) \to \mathit{arr}\,A\,B \to \mathit{arr}\,A\,B \to \mathit{Prop} \\ \quad \mathit{idArr} \in (A \in \mathit{obj}) \to \mathit{arr}\,A\,A \\ \quad \mathit{compArr} \in (A, B, C \in \mathit{obj}) \to \mathit{arr}\,A\,B \to \mathit{arr}\,B\,C \to \mathit{arr}\,A\,C \end{array}$$

In the rest of the chapter, *lt* will be represented by $\subseteq$, *compArr* by $\bullet$ and *idArr* by I. We do this in Alfa by indicating to the proof assistant to use those presentation aliases instead of the field names above. Since such choices of presentation are not done, as mentioned before, with Alfa commands but via menu options and clicking, they are not represented in the proof code. The reader should keep in mind this fact to read the proof code we present in our proofs.

4.3 Equality on arrows

The usual equality on arrows is now a derived notion. It is defined as the double inclusion of each arrow with respect to the other.

Definition

$$= \begin{pmatrix} f \in C.arr\,A\,B \\ g \in C.arr\,A\,B \end{pmatrix} \in Prop \qquad \begin{pmatrix} C \in LargePrecat \\ A \in C.obj \\ B \in C.obj \end{pmatrix}$$

$$f = g \equiv \begin{array}{l} \textbf{open}\ C\ \textbf{use}\ \subseteq \\ \textbf{in}\ (f \subseteq g)\ \&\ (g \subseteq f) \end{array}$$

4.4 LT-categories

Explanation

A *large LT-category* is a precategory that satisfies all the usual laws of a category, but with respect to their inequational versions. That is, each usual equation is split into the two corresponding inequations. However, making use of the derived notion of equality which we just presented, we can group the inequations and provide a presentation just like the standard one based on equality.

Besides the laws for identity arrows and associativity of composition of arrows, an LT-category must also satisfy that the inequality on arrows is a reflexive and transitive relation, and that composition is monotonic with regard to the inequality on arrows.

Definition

$LargeLTCat \ \in \#2$

$$\begin{array}{l}
LargeLTCat \equiv \\
\quad \textbf{sig} \\
\quad\quad pcat \in LargePrecat \\
\quad\quad \textbf{open}\ pcat\ \textbf{use}\ obj, arr, \subseteq, I, \bullet \\
\quad\quad ltIsRef \in (A, B \in obj, f \in arr\,A\,B) \to f \subseteq f \\
\quad\quad ltIsTra \in (A, B \in obj, f, g, h \in arr\,A\,B) \to \\
\quad\quad\quad (f \subseteq g) \to (g \subseteq h) \to f \subseteq h \\
\quad\quad compIsMonot \in (A, B, C \in obj, f1, f2 \in arr\,A\,B, g1, g2 \in arr\,B\,C) \to \\
\quad\quad\quad (f1 \subseteq f2) \to (g1 \subseteq g2) \to (f1 \bullet g1) \subseteq (f2 \bullet g2) \\
\quad\quad isLId \in (A, B \in obj, f \in arr\,A\,B) \to (I \bullet f) = f \\
\quad\quad isRId \in (A, B \in obj, f \in arr\,A\,B) \to (f \bullet I) = f \\
\quad\quad isAso \in (A, B, C, D \in obj, f \in arr\,A\,B, g \in arr\,B\,C, h \in arr\,C\,D) \to \\
\quad\quad\quad ((f \bullet g) \bullet h) = (f \bullet (g \bullet h))
\end{array}$$

4.4.1 Equality is an equivalence

Proposition

For any pair of objects A, B in an LT-category, equality on arrows between A and B is an equivalence relation.

Proof

$$eqIsRefl \in R = R \qquad \left(\begin{array}{l} A \in obj \\ B \in obj \\ R \in arr\,A\,B \end{array}\right)$$

$$eqIsRefl \equiv \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv ltIsRef\,R \\ snd \equiv ltIsRef\,R \end{array}\right.\end{array}$$

$$eqIsSymm \in (R = S) \rightarrow S = R \qquad \left(\begin{array}{l} A \in obj \\ B \in obj \\ R \in arr\,A\,B \\ S \in arr\,A\,B \end{array}\right)$$

$$eqIsSymm \equiv \lambda\,h \rightarrow \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv h.snd \\ snd \equiv h.fst \end{array}\right.\end{array}$$

$$eqIsTrans \left(\begin{array}{l} R \in arr\,A\,B \\ S \in arr\,A\,B \\ T \in arr\,A\,B \end{array}\right) \in (R = S) \rightarrow (S = T) \rightarrow R = T \qquad \left(\begin{array}{l} A \in obj \\ B \in obj \end{array}\right)$$

$$eqIsTrans\,R\,S\,T \equiv \lambda\,h\,h' \rightarrow \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv ltIsTra\,R\,S\,T\,h.fst\,h'.fst \\ snd \equiv ltIsTra\,T\,S\,R\,h'.snd\,h.snd \end{array}\right.\end{array}$$

4.4.2 About E-categories

The usual presentation of the definition for category is based on equality on arrows as a primitive concept. In our constructive setting, we have the notion of E-category [47] corresponding to that kind of presentation. The connection with LT-categories is quite obvious and intuitive: we can always get an E-category from an LT-category by taking as the equality on arrows the (defined) equality on arrows of the LT-category, that is double inclusion of arrows. We also need to show that composition preserves equality, but this follows easily from the fact that composition preserves the preorder of the LT-category we start with. For our purposes, it's both more natural and more efficient to start from LT-categories, since otherwise we would need to postulate a separate operation $\subseteq$ for arrows along with axioms that make its behaviour compatible with that of the equality of the E-category.

4.5 LT-allegories

Explanation

An *LT-allegory* is an LT-category with two additional operations, called *meet* and *converse*. Meet must be the greatest lower bound of its arguments. Converse must be involutive, order-preserving, and contravariant with regard to composition. Finally, there is a law tying together the behaviour of all the existing operations on arrows, called the *modular law*.

Definition

$LTAllegory \in \#2$

$$
LTAllegory \equiv
\begin{array}{l}
\textbf{sig} \\
\quad ltcat \in LargeLTCat \\
\quad \textbf{open}\ ltcat\ \textbf{use}\ pcat \\
\quad \textbf{open}\ pcat\ \textbf{use}\ obj, arr, \subseteq, \bullet \\
\quad meet \in (A, B \in obj, f, g \in arr\,A\,B) \to arr\,A\,B \\
\quad conv \in (A, B \in obj, f \in arr\,A\,B) \to arr\,B\,A \\
\quad defMeet \in (A, B \in obj, f, g, x \in arr\,A\,B) \to \\
\qquad (x \subseteq (f \cap g)) \leftrightarrow ((x \subseteq f)\ \&\ (x \subseteq g)) \\
\quad isInvolConv \in (A, B \in obj, f \in arr\,A\,B) \to ((f^\circ)^\circ) = f \\
\quad isOrdPresConv \in (A, B \in obj, f, g \in arr\,A\,B) \to \\
\qquad (f \subseteq g) \leftrightarrow ((f^\circ) \subseteq (g^\circ)) \\
\quad isContravConv \in (A, B, C \in obj, f \in arr\,A\,B, g \in arr\,B\,C) \to \\
\qquad ((f \bullet g)^\circ) = ((g^\circ) \bullet (f^\circ)) \\
\quad modLaw \in (A, B, C \in obj, f \in arr\,A\,B, g \in arr\,B\,C, h \in arr\,A\,C) \to \\
\qquad ((f \bullet g) \cap h) \subseteq ((f \cap (h \bullet (g^\circ))) \bullet g)
\end{array}
$$

Some useful lemmas

We now prove a few simple lemmas which hold for arbitrary LT-allegories. The lemmas are not so interesting by themselves, but they provide useful abbreviations in numerous (and more important) proofs to come later.

Lemma

Any relation contained in the meet of two other relations is also contained in each of other two relations.

$$fstLTMeet \in (X \subseteq (R \cap S)) \to X \subseteq R \qquad \left(\begin{array}{l} A \in obj \\ B \in obj \\ R \in arr\,A\,B \\ S \in arr\,A\,B \\ X \in arr\,A\,B \end{array}\right)$$

$$fstLTMeet \equiv \lambda\, h \to ((defMeet\,R\,S\,X)\,.fst\,h)\,.fst$$

$$sndLTMeet \in (X \subseteq (R \cap S)) \to X \subseteq S \qquad \left(\begin{array}{l} A \in obj \\ B \in obj \\ R \in arr\,A\,B \\ S \in arr\,A\,B \\ X \in arr\,A\,B \end{array}\right)$$

$$sndLTMeet \equiv \lambda\, h \to ((defMeet\,R\,S\,X)\,.fst\,h)\,.snd$$

Lemma

Any relation contained in two other relations is also contained in the meet of those two other relations.

$$mixLTMeet \in (X \subseteq R) \to (X \subseteq S) \to X \subseteq (R \cap S) \qquad \left(\begin{array}{l} A \in obj \\ B \in obj \\ R \in arr\,A\,B \\ S \in arr\,A\,B \\ X \in arr\,A\,B \end{array}\right)$$

$$mixLTMeet \equiv \lambda\, hR\, hS \to (defMeet\,R\,S\,X)\,.snd \left(\begin{array}{l} \textbf{struct} \\ \quad \left[\begin{array}{l} fst \equiv hR \\ snd \equiv hS \end{array}\right. \end{array}\right)$$

Lemma

To prove $R = S$, just prove that any relation is contained in R if and only if it is contained in S.

$$indirProof \in (X \in arr\,A\,B) \to ((X \subseteq R) \leftrightarrow (X \subseteq S)) \to R = S \qquad \left(\begin{array}{l} A \in obj \\ B \in obj \\ R \in arr\,A\,B \\ S \in arr\,A\,B \end{array}\right)$$

$$indirProof \equiv \lambda\, h\ \rightarrow\ \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv (h\,R)\,.fst\ (ltIsRef\,R)\\ snd \equiv (h\,S)\,.snd\ (ltIsRef\,S)\end{array}\right.\end{array}$$

Lemma

If an equation $R = S$ holds, then any relation is contained in R if and only if it is contained in S.

$$getLTFromEq\ \in (R = S)\ \rightarrow\ (X\ \in\ arr\,A\,B) \rightarrow\ (X\ \subseteq\ R)\ \leftrightarrow\ (X\ \subseteq\ S)$$
$$\left(\begin{array}{l} A \in obj\\ B \in obj\\ R \in arr\,A\,B\\ S \in arr\,A\,B\end{array}\right)$$

$$getLTFromEq \equiv \lambda\, h\, X\ \rightarrow\ \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv \lambda\, h'\ \rightarrow\ ltIsTra\,X\,R\,S\,h'\,h.fst\\ snd \equiv \lambda\, h'\ \rightarrow\ ltIsTra\,X\,S\,R\,h'\,h.snd\end{array}\right.\end{array}$$

Properties of LT-allegories

We will now present some properties verified by all LT-allegories, along with their formal proofs.

Proposition

The meet of two relations is contained in each of the two relations.

$$meetInArgL \left(\begin{array}{l} R \in arr\,A\,B\\ S \in arr\,A\,B\end{array}\right) \in (R\ \cap\ S)\ \subseteq\ R \qquad \left(\begin{array}{l} A \in obj\\ B \in obj\end{array}\right)$$

$$meetInArgL\,R\,S \equiv fstLTMeet\ (ltIsRef\ (R\ \cap\ S))$$

$$meetInArgR \left(\begin{array}{l} R \in arr\,A\,B\\ S \in arr\,A\,B\end{array}\right) \in (R\ \cap\ S)\ \subseteq\ S \qquad \left(\begin{array}{l} A \in obj\\ B \in obj\end{array}\right)$$

$$meetInArgR\,R\,S \equiv sndLTMeet\ (ltIsRef\ (R\ \cap\ S))$$

Proposition

For any two objects A,B, the meet of arrows between A and B is a commutative operation.

$$isCommMeet \ \in (R \cap S) = (S \cap R) \qquad \begin{pmatrix} A \in obj \\ B \in obj \\ R \in arr\,A\,B \\ S \in arr\,A\,B \end{pmatrix}$$

$$\begin{array}{l} isCommMeet \equiv indirProof\,\lambda\,X \ \rightarrow \\ \quad \textbf{struct} \\ \quad \left[\begin{array}{l} fst \equiv \lambda\,h \ \rightarrow \ mixLTMeet \ (sndLTMeet\,h) \ (fstLTMeet\,h) \\ snd \equiv \lambda\,h \ \rightarrow \ mixLTMeet \ (sndLTMeet\,h) \ (fstLTMeet\,h) \end{array} \right. \end{array}$$

Proposition

For any two objects A,B, the meet of arrows between A and B is an idempotent operation.

$$isIdempMeet \ \in (R \cap R) = R \qquad \begin{pmatrix} A \in obj \\ B \in obj \\ R \in arr\,A\,B \end{pmatrix}$$

$$isIdempMeet \equiv indirProof\,\lambda\,X \ \rightarrow \quad \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv \lambda\,h \ \rightarrow \ fstLTMeet\,h \\ snd \equiv \lambda\,h \ \rightarrow \ mixLTMeet\,h\,h \end{array} \right. \end{array}$$

Proposition

For any two objects A,B, the meet of arrows between A and B is an associative operation.

$$isAssocMeet \ \in ((R \cap S) \cap T) = (R \cap (S \cap T)) \qquad \begin{pmatrix} A \in obj \\ B \in obj \\ R \in arr\,A\,B \\ S \in arr\,A\,B \\ T \in arr\,A\,B \end{pmatrix}$$

$$isAssocMeet \equiv indirProof\,\lambda\,X \;\rightarrow$$
$$\begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv \lambda\,h \;\rightarrow\; mixLTMeet\;(fstLTMeet\;(fstLTMeet\,h))\\ \quad (mixLTMeet\;(sndLTMeet\;(fstLTMeet\,h))\;(sndLTMeet\,h))\end{array}\right.\\ \left[\begin{array}{l} snd \equiv \lambda\,h \;\rightarrow\; mixLTMeet\\ \quad (mixLTMeet\;(fstLTMeet\,h)\;(fstLTMeet\;(sndLTMeet\,h)))\\ \quad (sndLTMeet\;(sndLTMeet\,h))\end{array}\right.\end{array}$$

Proposition

$R \subseteq S$ is equivalent to $R \cap S = R$.

$$ltAsMeet \;\in ((R \;\cap\; S) = R) \;\leftrightarrow\; (R \;\subseteq\; S) \qquad \left(\begin{array}{l} A \in obj\\ B \in obj\\ R \in arr\,A\,B\\ S \in arr\,A\,B\end{array}\right)$$

$$ltAsMeet \equiv \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv \lambda\,h \;\rightarrow\; sndLTMeet\,h.snd\end{array}\right.\\ \left[\begin{array}{ll} snd \equiv \lambda\,h \;\rightarrow & \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv meetInArgL\,R\,S\\ snd \equiv mixLTMeet\;(ltIsRef\,R)\;h\end{array}\right.\end{array}\end{array}\right.\end{array}$$

Proposition

Converse distributes over meet.

$$distrConvMeet \;\in ((R \;\cap\; S)^{\circ}) = ((R^{\circ}) \;\cap\; (S^{\circ})) \qquad \left(\begin{array}{l} A \in obj\\ B \in obj\\ R \in arr\,A\,B\\ S \in arr\,A\,B\end{array}\right)$$

$$distrConvMeet \equiv indirProof\,\lambda\,X \;\rightarrow$$

struct

$$\left[\begin{array}{l} fst \equiv \lambda\, h \;\rightarrow\; \begin{array}{l} \textbf{let} \left|\begin{array}{l} lem \in (X^{\circ}) \subseteq (R \cap S) \\ lem \equiv ltIsTra\ (X^{\circ}) \left(((R \cap S)^{\circ})^{\circ}\right) (R \cap S) \\ \qquad ((isOrdPresConv\ X\ ((R \cap S)^{\circ})).fst\ h) \\ \qquad (isInvolConv\ (R \cap S)).fst \end{array}\right. \\ \textbf{in}\ mixLTMeet \\ \quad (ltIsTra\ X\ ((X^{\circ})^{\circ})\ (R^{\circ})\ (isInvolConv\ X).snd \\ \qquad ((isOrdPresConv\ (X^{\circ})\ R).fst\ (fstLTMeet\ lem))) \\ \quad (ltIsTra\ X\ ((X^{\circ})^{\circ})\ (S^{\circ})\ (isInvolConv\ X).snd \\ \qquad ((isOrdPresConv\ (X^{\circ})\ S).fst\ (sndLTMeet\ lem))) \end{array} \end{array}\right.$$

$$\left[\begin{array}{l} snd \equiv \lambda\, h \;\rightarrow\; ltIsTra\ X\ ((X^{\circ})^{\circ})\ ((R \cap S)^{\circ})\ (isInvolConv\ X).snd \\ \quad ((isOrdPresConv\ (X^{\circ})\ (R \cap S)).fst\ (mixLTMeet \\ \qquad (ltIsTra\ (X^{\circ})\ ((R^{\circ})^{\circ})\ R \\ \qquad\quad ((isOrdPresConv\ X\ (R^{\circ})).fst\ (fstLTMeet\ h))\ (isInvolConv\ R).fst) \\ \qquad (ltIsTra\ (X^{\circ})\ ((S^{\circ})^{\circ})\ S \\ \qquad\quad ((isOrdPresConv\ X\ (S^{\circ})).fst\ (sndLTMeet\ h))\ (isInvolConv\ S).fst))) \end{array}\right.$$

The proof is carried in the indirect style way, that is by proving that something is included in the left hand side iff it is included in the right hand side. Each direction of this indirect proof is carried by a small chain of inequalities, in which the converses are shifted in and out of the meet's arguments as necessary by using the properties of converse from the definition of LT-allegory.

Proposition

Composition subdistributes over meet, both from left and right.

$$isSubdistrLCompMeet\ \in (R \bullet (S \cap T)) \subseteq ((R \bullet S) \cap (R \bullet T)) \qquad \left(\begin{array}{l} A \in obj \\ B \in obj \\ C \in obj \\ R \in arr\ A\ B \\ S \in arr\ B\ C \\ T \in arr\ B\ C \end{array}\right)$$

$$isSubdistrLCompMeet \equiv \begin{array}{l} mixLTMeet \\ (compIsMonot\ R\ R\ (S \cap T)\ S\ (ltIsRef\ R)\ (meetInArgL\ S\ T)) \\ (compIsMonot\ R\ R\ (S \cap T)\ T\ (ltIsRef\ R)\ (meetInArgR\ S\ T)) \end{array}$$

$$isSubdistrRCompMeet\ \in ((R \cap S) \bullet T) \subseteq ((R \bullet T) \cap (S \bullet T)) \qquad \left(\begin{array}{l} A \in obj \\ B \in obj \\ C \in obj \\ R \in arr\ A\ B \\ S \in arr\ A\ B \\ T \in arr\ B\ C \end{array}\right)$$

$$isSubdistrRCompMeet \equiv \begin{array}{l} mixLTMeet \\ (compIsMonot\ (R \cap S)\ R\,T\,T\ (meetInArgL\,R\,S)\ (ltIsRef\,T)) \\ (compIsMonot\ (R \cap S)\ S\,T\,T\ (meetInArgR\,R\,S)\ (ltIsRef\,T)) \end{array}$$

Proposition

Equality of arrows is a congruence for composition.

$$substComp \begin{pmatrix} R1 \in arr\,A\,B \\ R2 \in arr\,A\,B \\ S1 \in arr\,B\,C \\ S2 \in arr\,B\,C \end{pmatrix} \in (R1 = R2) \rightarrow (S1 = S2) \rightarrow (R1 \bullet S1) = (R2 \bullet S2)$$
$$\begin{pmatrix} A \in obj \\ B \in obj \\ C \in obj \end{pmatrix}$$

$$\begin{array}{l} substComp\ R1\ R2\ S1\ S2 \equiv \lambda\, hR\, hS \rightarrow \\ \quad \textbf{struct} \\ \quad \left[\begin{array}{l} fst \equiv compIsMonot\ R1\ R2\ S1\ S2\ hR.fst\ hS.fst \\ snd \equiv compIsMonot\ R2\ R1\ S2\ S1\ hR.snd\ hS.snd \end{array} \right. \end{array}$$

Proposition

Equality of arrows is a congruence for converse.

$$substConv \begin{pmatrix} R \in arr\,A\,B \\ S \in arr\,A\,B \end{pmatrix} \in (R = S) \rightarrow (R^{\circ}) = (S^{\circ}) \qquad \begin{pmatrix} A \in obj \\ B \in obj \end{pmatrix}$$

$$substConv\ R\ S \equiv \lambda\, h \rightarrow \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv (isOrdPresConv\ R\ S).fst\ h.fst \\ snd \equiv (isOrdPresConv\ S\ R).fst\ h.snd \end{array} \right. \end{array}$$

Proposition

The identity arrow is its own converse.

$$idIsOwnConv \in (I^{\circ}) = I \qquad (A \in obj)$$

$$
\begin{array}{ll}
 & \mathit{eqIsTrans}\,(I^\circ)\,((I^\circ)\ \bullet\ I)\,I\,(\mathit{eqIsSymm}\,(\mathit{isRId}\,(I^\circ))) \\
 & \quad (\mathit{eqIsTrans}\,((I^\circ)\ \bullet\ I)\,((I^\circ)\ \bullet\ ((I^\circ)^\circ))\,I \\
 & \quad\quad (\mathit{substComp}\,(I^\circ)\,(I^\circ)\,I\,((I^\circ)^\circ) \\
 & \quad\quad\quad \mathit{eqIsRefl}\,(\mathit{eqIsSymm}\,(\mathit{isInvolConv}\,I))) \\
\mathit{idIsOwnConv} \equiv & \quad\quad\quad (\mathit{eqIsTrans}\,((I^\circ)\ \bullet\ ((I^\circ)^\circ))\,(((I^\circ)\ \bullet\ I)^\circ)\,I \\
 & \quad\quad\quad\quad (\mathit{eqIsSymm}\,(\mathit{isContravConv}\,(I^\circ)\,I)) \\
 & \quad\quad\quad\quad (\mathit{eqIsTrans}\,(((I^\circ)\ \bullet\ I)^\circ)\,((I^\circ)^\circ)\,I \\
 & \quad\quad\quad\quad\quad (\mathit{substConv}\,((I^\circ)\ \bullet\ I)\,(I^\circ)\,(\mathit{isRId}\,(I^\circ))) \\
 & \quad\quad\quad\quad\quad (\mathit{isInvolConv}\,I))))
\end{array}
$$

The structure of the proof is a chain of equalities that can be followed by looking at the *eqIsTrans* calls. The idea is to use all the properties of converse in the context of a composition of identity relations or their converses. These compositions allow us to introduce and remove identity relations along the chain, in such a way that we can move the converse applications until they can be removed and produce a single identity relation without converses.

4.6 Distributive LT-allegories

A *distributive LT-allegory* is an LT-allegory that adds a new operation, *join*, and a new constant relation *zero*. Join is defined as the lowest upper bound of its arguments. Both meet and composition distribute over join to the left. Also, zero is indeed a zero for meet and composition, besides being a neutral element for join.

Definition

$\mathit{DistribAllegory}\ \in \#2$

$\mathit{DistribAllegory} \equiv$

sig
$algy \in LTAllegory$
open $algy$ **use** $ltcat, \cap$
open $ltcat$ **use** $pcat$
open $pcat$ **use** $obj, arr, \subseteq, \bullet$
$join \in (A, B \in obj, f, g \in arr\,A\,B) \to arr\,A\,B$
$zero \in (A, B \in obj) \to arr\,A\,B$
$defJoin \in (A, B \in obj, f, g, x \in arr\,A\,B) \to$
$((f \cup g) \subseteq x) \leftrightarrow ((f \subseteq x)\ \&\ (g \subseteq x))$
$isDistrMeetJoin \in (A, B \in obj, f, g, h \in arr\,A\,B) \to$
$(f \cap (g \cup h)) = ((f \cap g) \cup (f \cap h))$
$isDistrComposJoin \in (A, B, C \in obj, f \in arr\,A\,B, g, h \in arr\,B\,C) \to$
$(f \bullet (g \cup h)) = ((f \bullet g) \cup (f \bullet h))$
$isZeroMeet \in (A, B \in obj, f \in arr\,A\,B) \to (f \cap \emptyset) = \emptyset$
$isZeroJoin \in (A, B \in obj, f \in arr\,A\,B) \to (f \cup \emptyset) = f$
$isZeroCompos \in (A, B, C \in obj, f \in arr\,A\,B) \to (f \bullet \emptyset) = \emptyset$

Comments

The first thing we should call attention to is that *DistribAllegory* is an extension of the signature *LTAllegory*. This is done by having a field inside the new signature whose type is precisely *LTAllegory*. Alfa/Agda lacks a record subtyping feature, so this is how we perform such extensions in our proof code. The second thing we call attention to is the use of successive **open** sentences inside the signature definition. Each **open** makes available the field names of the signature, and if one of those fields is in turn a signature, it can also be opened. This is in fact what we do, to avoid lengthy chains of dot-notation selections that would obscure the definition quite badly.

On locally complete allegories

In Bird and de Moor's book the introduction of join and zero is not done in the way we have done it here, but instead by means of the concept of locally complete allegories. In these, join is defined as an operation taking as argument a potentially infinte number of arrows of the same type $\mathcal{H}$, such that $\cup\mathcal{H} \subseteq X$ iff $\forall R \in \mathcal{H}.R \subseteq X$. To define this in type theory, we would have $\mathcal{H} \in \mathrm{Hom}(A, B) \to Set$. The definition of the concrete join would be that $a(\cup\mathcal{H})b$ iff $\exists R \in \mathrm{Hom}(A, B).\ \mathcal{H}(R)\ \&\ aRb$. However, the existential wouldn't be in *Set* but in *Type*, making $\cup\mathcal{H}$ break the typing of a propositional relation. For this reason we will work with distributive allegories instead, since in their case the definition of concrete join does have the proper type for a propositional relation.

Some useful lemmas

Lemma

If the join of two relations is contained in another relation, then each of the joined relations is also contained in that other relation.

$$fstLTJoin \in ((R \cup S) \subseteq X) \to R \subseteq X \qquad \left(\begin{array}{l} A \in obj \\ B \in obj \\ R \in arr\,A\,B \\ S \in arr\,A\,B \\ X \in arr\,A\,B \end{array}\right)$$

$$fstLTJoin \equiv \lambda\, h \to ((defJoin\,R\,S\,X).fst\,h).fst$$

$$sndLTJoin \in ((R \cup S) \subseteq X) \to S \subseteq X \qquad \left(\begin{array}{l} A \in obj \\ B \in obj \\ R \in arr\,A\,B \\ S \in arr\,A\,B \\ X \in arr\,A\,B \end{array}\right)$$

$$sndLTJoin \equiv \lambda\, h \to ((defJoin\,R\,S\,X).fst\,h).snd$$

Lemma

If two relations are contained in a third one, then their join is also contained in the third relation.

$$mixLTJoin \in (R \subseteq X) \to (S \subseteq X) \to (R \cup S) \subseteq X \qquad \left(\begin{array}{l} A \in obj \\ B \in obj \\ R \in arr\,A\,B \\ S \in arr\,A\,B \\ X \in arr\,A\,B \end{array}\right)$$

$$mixLTJoin \equiv \lambda\, hR\, hS \to (defJoin\,R\,S\,X).snd \left(\begin{array}{l} \textbf{struct} \\ \quad \left[\begin{array}{l} fst \equiv hR \\ snd \equiv hS \end{array}\right. \end{array}\right)$$

Properties of distributive LT-allegories

Proposition

For any two relations that are joined, each relation is contained in their join.

$$argLinJoin \begin{pmatrix} R \in arr\ A\ B \\ S \in arr\ A\ B \end{pmatrix} \in R \subseteq (R \cup S) \qquad \begin{pmatrix} A \in obj \\ B \in obj \end{pmatrix}$$

$$argLinJoin\ R\ S \equiv fstLTJoin\ (ltIsRef\ (R \cup S))$$

$$argRinJoin \begin{pmatrix} R \in arr\ A\ B \\ S \in arr\ A\ B \end{pmatrix} \in S \subseteq (R \cup S) \qquad \begin{pmatrix} A \in obj \\ B \in obj \end{pmatrix}$$

$$argRinJoin\ R\ S \equiv sndLTJoin\ (ltIsRef\ (R \cup S))$$

Proposition

Join is monotonic with respect to arrow inclusion.

$$isMonotJoin \in (R1 \subseteq R2) \to (S1 \subseteq S2) \to (R1 \cup S1) \subseteq (R2 \cup S2)$$
$$\begin{pmatrix} A \in obj \\ B \in obj \\ R1 \in arr\ A\ B \\ R2 \in arr\ A\ B \\ S1 \in arr\ A\ B \\ S2 \in arr\ A\ B \end{pmatrix}$$

$$isMonotJoin \equiv \begin{array}{l} \lambda\, hR\, hS \to mixLTJoin \\ \quad (ltIsTra\ R1\ R2\ (R2 \cup S2)\ hR\ (argLinJoin\ R2\ S2)) \\ \quad (ltIsTra\ S1\ S2\ (R2 \cup S2)\ hS\ (argRinJoin\ R2\ S2)) \end{array}$$

Proposition

For any two objects A,B, the join of two arrows between A and B is a commutative operation.

$$isCommJoin \in (R \cup S) = (S \cup R) \qquad \begin{pmatrix} A \in obj \\ B \in obj \\ R \in arr\ A\ B \\ S \in arr\ A\ B \end{pmatrix}$$

$$isCommJoin \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv mixLTJoin\ (argRinJoin\ S\ R)\ (argLinJoin\ S\ R) \\ snd \equiv mixLTJoin\ (argRinJoin\ R\ S)\ (argLinJoin\ R\ S) \end{array} \right. \end{array}$$

Proposition

For any two objects A,B, join of arrows between A and B is an idempotent operation.

$$isIdempJoin \in (R \cup R) = R \qquad \left(\begin{array}{l} A \in obj \\ B \in obj \\ R \in arr\,A\,B \end{array}\right)$$

$$isIdempJoin \equiv \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv mixLTJoin\ (ltIsRef\,R)\ (ltIsRef\,R) \\ snd \equiv fstLTJoin\ (ltIsRef\ (R \cup R)) \end{array}\right.\end{array}$$

Proposition

For any two objects A,B, join of arrows between A and B is an associative operation.

$$isAssocJoin \in ((R \cup S) \cup T) = (R \cup (S \cup T)) \qquad \left(\begin{array}{l} A \in obj \\ B \in obj \\ R \in arr\,A\,B \\ S \in arr\,A\,B \\ T \in arr\,A\,B \end{array}\right)$$

$$isAssocJoin \equiv$$

$$\begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv \\ \quad mixLTJoin \\ \quad\left(\begin{array}{l} mixLTJoin \\ \quad (argLinJoin\,R\ (S \cup T)) \\ \quad (ltIsTra\,S\ (S \cup T)\ (R \cup (S \cup T)) \\ \qquad (argLinJoin\,S\,T)\ (argRinJoin\,R\ (S \cup T))) \end{array}\right) \\ \quad (ltIsTra\,T\ (S \cup T)\ (R \cup (S \cup T)) \\ \qquad (argRinJoin\,S\,T)\ (argRinJoin\,R\ (S \cup T))) \end{array}\right. \\ \left[\begin{array}{l} snd \equiv \\ \quad mixLTJoin \\ \quad (ltIsTra\,R\,(R \cup S)\ ((R \cup S) \cup T) \\ \qquad (argLinJoin\,R\,S)\ (argLinJoin\ (R \cup S)\ T)) \\ \quad\left(\begin{array}{l} mixLTJoin \\ \quad (ltIsTra\,S\ (R \cup S)\ ((R \cup S) \cup T) \\ \qquad (argRinJoin\,R\,S)\ (argLinJoin\ (R \cup S)\ T)) \\ \quad (argRinJoin\ (R \cup S)\ T) \end{array}\right) \end{array}\right.\end{array}$$

Proposition

Converse distributes over join.

$$distConvJoin \in ((R \cup S)^\circ) = ((R^\circ) \cup (S^\circ)) \qquad \begin{pmatrix} A \in obj \\ B \in obj \\ R \in arr\,A\,B \\ S \in arr\,A\,B \end{pmatrix}$$

$distConvJoin \equiv$

$$\begin{array}{l}
\textbf{struct} \\
\left[\begin{array}{l}
fst \equiv \\
\quad ltIsTra \\
\qquad ((R \cup S)^\circ) \\
\qquad \big((((R^\circ) \cup (S^\circ))^\circ)^\circ\big) \\
\qquad ((R^\circ) \cup (S^\circ)) \\
\qquad \left(\begin{array}{l}
(isOrdPresConv\ (R \cup S)\ (((R^\circ) \cup (S^\circ))^\circ)).fst \\
\left(\begin{array}{l}
mixLTJoin \\
\quad ((isOrdPresConv\,R\ (((R^\circ) \cup (S^\circ))^\circ)).snd \\
\qquad (ltIsTra\ (R^\circ)\ ((R^\circ) \cup (S^\circ))\ \big((((R^\circ) \cup (S^\circ))^\circ)^\circ\big) \\
\qquad\quad (argLinJoin\ (R^\circ)\ (S^\circ)) \\
\qquad\quad (isInvolConv\ ((R^\circ) \cup (S^\circ))).snd)) \\
\quad ((isOrdPresConv\,S\ (((R^\circ) \cup (S^\circ))^\circ)).snd \\
\qquad (ltIsTra\ (S^\circ)\ ((R^\circ) \cup (S^\circ))\ \big((((R^\circ) \cup (S^\circ))^\circ)^\circ\big) \\
\qquad\quad (argRinJoin\ (R^\circ)\ (S^\circ)) \\
\qquad\quad (isInvolConv\ ((R^\circ) \cup (S^\circ))).snd))
\end{array}\right)
\end{array}\right) \\
\qquad (isInvolConv\ ((R^\circ) \cup (S^\circ))).fst
\end{array}\right. \\
\left[\begin{array}{l}
snd \equiv mixLTJoin\ ((isOrdPresConv\,R\ (R \cup S)).fst\ (argLinJoin\,R\,S)) \\
\qquad ((isOrdPresConv\,S\ (R \cup S)).fst\ (argRinJoin\,R\,S))
\end{array}\right.
\end{array}$$

The idea behind this proof is that in the left to right direction we use the small inequality chain indicated by *ltIsTra*, and in the right to left direction we can more easily just use the fact that converse is order preserving.

Proposition

The zero relation between A and B is contained in every relation of the same type.

$$zeroIsMinim\ (R \in arr\,A\,B) \in \emptyset \subseteq R \qquad \begin{pmatrix} A \in obj \\ B \in obj \end{pmatrix}$$

$$zeroIsMinim\ R \equiv ltAsMeet.fst \left(\begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv ltIsTra\ (\emptyset \cap R)\ (R \cap \emptyset)\ \emptyset \\ \quad isCommMeet.fst\ (meetInArgR\ R\ \emptyset) \\ snd \equiv ltIsTra\ \emptyset\ (R \cap \emptyset)\ (\emptyset \cap R) \\ \quad (isZeroMeet\ R)\ .snd\ isCommMeet.fst \end{array} \right. \end{array} \right)$$

Proposition

The zero relation is its own converse.

$$zeroConvZero\ \in (\emptyset^\circ) = \emptyset \qquad \left(\begin{array}{l} A \in obj \\ B \in obj \end{array} \right)$$

$$zeroConvZero \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv (isOrdPresConv\ (\emptyset^\circ)\ \emptyset)\ .snd\ (ltIsTra\ ((\emptyset^\circ)^\circ)\ \emptyset\ (\emptyset^\circ) \\ \quad (isInvolConv\ \emptyset)\ .fst\ (zeroIsMinim\ (\emptyset^\circ))) \\ snd \equiv zeroIsMinim\ (\emptyset^\circ) \end{array} \right. \end{array}$$

Proposition

The zero relation is also a left zero for composition.

$$isZeroCompos'\ \in (\emptyset \bullet R) = \emptyset \qquad \left(\begin{array}{l} A \in obj \\ B \in obj \\ C \in obj \\ R \in arr\ B\ C \end{array} \right)$$

$$isZeroCompos' \equiv$$

$$
\textbf{struct}\left[\begin{array}{l}
\mathit{fst} \equiv \\
\quad (\mathit{isOrdPresConv}\ (\emptyset \bullet R)\ \emptyset)\,.\mathit{snd} \\
\quad \left(\begin{array}{l}
\mathit{ltIsTra} \\
\quad ((\emptyset \bullet R)^{\circ}) \\
\quad ((R^{\circ}) \bullet (\emptyset^{\circ})) \\
\quad (\emptyset^{\circ}) \\
\quad (\mathit{isContravConv}\ \emptyset\ R)\,.\mathit{fst} \\
\quad \left(\begin{array}{l}
\mathit{ltIsTra} \\
\quad ((R^{\circ}) \bullet (\emptyset^{\circ})) \\
\quad ((R^{\circ}) \bullet \emptyset) \\
\quad (\emptyset^{\circ}) \\
\quad (\mathit{compIsMonot}\ (R^{\circ})\ (R^{\circ})\ (\emptyset^{\circ})\ \emptyset\ (\mathit{ltIsRef}\ (R^{\circ})) \\
\qquad\qquad \mathit{zeroConvZero}.\mathit{fst}) \\
\quad (\mathit{ltIsTra}\ ((R^{\circ}) \bullet \emptyset)\ \emptyset\ (\emptyset^{\circ}) \\
\qquad (\mathit{isZeroCompos}\ (R^{\circ}))\,.\mathit{fst}\ \mathit{zeroConvZero}.\mathit{snd})
\end{array}\right)
\end{array}\right) \\
\mathit{snd} \equiv \mathit{zeroIsMinim}\ (\emptyset \bullet R)
\end{array}\right.
$$

The idea behind this proof is that in the left to right direction of the doble inclusion we need to use contravariance of the converse, so we can move the empty relation to the right side of the composition, where the existing property of it being a right zero can finally be applied.

Proposition

Composition also distributes over join on the right.

$$
\mathit{isDistrComposJoin}' \in ((R \cup S) \bullet T) = ((R \bullet T) \cup (S \bullet T)) \qquad \left(\begin{array}{l}
A \in \mathit{obj} \\
B \in \mathit{obj} \\
C \in \mathit{obj} \\
R \in \mathit{arr}\ A\ B \\
S \in \mathit{arr}\ A\ B \\
T \in \mathit{arr}\ B\ C
\end{array}\right)
$$

$\mathit{isDistrComposJoin}' \equiv$

```
struct
  fst ≡
    ltIsTra
      ((R ∪ S) • T)
      ((((R ∪ S) • T)°)°)
      ((R • T) ∪ (S • T))
      (isInvolConv ((R ∪ S) • T)) .snd
      ( ltIsTra
          ((((R ∪ S) • T)°)°)
          (((T°) • ((R ∪ S)°))°)
          ((R • T) ∪ (S • T))
          ((isOrdPresConv (((R ∪ S) • T)°) ((T°) • ((R ∪ S)°))) .fst
              (isContravConv (R ∪ S) T) .fst)
          ( ltIsTra
              (((T°) • ((R ∪ S)°))°)
              (((T°) • ((R°) ∪ (S°)))°)
              ((R • T) ∪ (S • T))
              ((isOrdPresConv ((T°) • ((R ∪ S)°)) ((T°) • ((R°) ∪ (S°)))) .fst
                  (compIsMonot (T°) (T°) ((R ∪ S)°) ((R°) ∪ (S°))
                      (ltIsRef (T°)) distConvJoin.fst))
              ( ltIsTra
                  (((T°) • ((R°) ∪ (S°)))°)
                  ((((T°) • (R°)) ∪ ((T°) • (S°)))°)
                  ((R • T) ∪ (S • T))
                  ((isOrdPresConv ((T°) • ((R°) ∪ (S°)))
                          (((T°) • (R°)) ∪ ((T°) • (S°)))).fst
                      (isDistrComposJoin C B A (T°) (R°) (S°)) .fst)
                  ( ltIsTra
                      ((((T°) • (R°)) ∪ ((T°) • (S°)))°)
                      ((((R • T)°) ∪ ((S • T)°))°)
                      ((R • T) ∪ (S • T))
                      ((isOrdPresConv (((T°) • (R°)) ∪ ((T°) • (S°)))
                              (((R • T)°) ∪ ((S • T)°))).fst
                          (isMonotJoin (isContravConv R T) .snd
                              (isContravConv S T) .snd))
                      ( ltIsTra
                          ((((R • T)°) ∪ ((S • T)°))°)
                          ((((R • T)°)°) ∪ (((S • T)°)°))
                          ((R • T) ∪ (S • T))
                          distConvJoin.fst
                          (isMonotJoin (isInvolConv (R • T)) .fst
                              (isInvolConv (S • T)) .fst) ) ) ) ) )
  snd ≡ mixLTJoin (compIsMonot R (R ∪ S) T T (argLinJoin R S) (ltIsRef T))
      (compIsMonot S (R ∪ S) T T (argRinJoin R S) (ltIsRef T))
```

The left to right direction of the above proof by double inclusion is quite complicated. It should be hopefully not too hard to figure out by following the chains of inequalities denoted by the *ltIsTra* calls. This is one case in which the proof is inherently (in)equational in nature, something for which Alfa is not too adequate at the moment. Each part of the transitive chain requires a new proof object, and it can be quite difficult to keep track of what one is doing during the completion of this kind of proof.

Comments

As the reader has probably noticed, the proof objects that make up the proofs of the several properties of allegories have become increasingly bigger. The reason for this is that they reflect proofs that consist of a chain of inequalities: that is, they are proofs by transitivity of the preorder between relations (arrows). At present there are no special features in Alfa/Agda for dealing with long chains of this kind of (in)equational reasoning. Each proof by transitivity contains the three things compared, in the order in which they fit the inequality chain, and the two proofs that establish the transitivity. It could be argued that argument-hiding the original two things and leaving just the bridging one along with the two proofs would shorten things considerably. However, we feel that the price in legibility is too high, as there is no more a clear annotation of the three pieces of the transitive chain to look at when trying to read the proofs that follow.

4.7 Special kinds of relations

4.7.1 Simple arrows

Simple relations are those that behave as partial functions, that is, for every element of the first set there is at most one element of the second set that is related to it. Here we use the equivalent relational definition, that R is simple iff $R^\circ \bullet R \subseteq I$.

Why is this the definition? If we unfold the meaning of the relational inequation in terms of concrete relations, we obtain precisely that $\forall x \in A.\forall y, y' \in B. x\,R\,y \;\wedge\; x\,R\,y' \to y =_B y'$. This is equivalent to the intuitive meaning of being a partial function.

Definition

$$\mathit{simpleArr}\ (R \in X.\mathit{ltcat.pcat.arr}\ A\ B) \in \mathit{Prop} \qquad \begin{pmatrix} X \in \mathit{LTAllegory} \\ A \in X.\mathit{ltcat.pcat.obj} \\ B \in X.\mathit{ltcat.pcat.obj} \end{pmatrix}$$

$$
\mathit{simpleArr}\ R \equiv \begin{array}{l} \textbf{open}\ X\ \textbf{use}\ \mathit{ltcat}, \circ \\ \textbf{in}\ \begin{array}{l} \textbf{open}\ \mathit{ltcat}\ \textbf{use}\ \mathit{pcat} \\ \textbf{in}\ \begin{array}{l} \textbf{open}\ \mathit{pcat}\ \textbf{use}\ \subseteq, I, \bullet \\ \textbf{in}\ ((R^{\circ}) \bullet R) \subseteq I \end{array} \end{array} \end{array}
$$

4.7.2 Entire arrows

Entire relations are those that are defined for all elements of the first set. That is, all elements of the first set are related to at least one element of the second set. Once again, we express this in terms of a relational definition, saying that R is entire iff $I \subseteq R \bullet R^{\circ}$. And also again, we can make sense of this relational inequation by unfolding it in terms of concrete relations, in which case we obtain $\forall x \in A. \exists y \in B. x\, R\, y$.

Definition

$$
\mathit{entireArr}\ (R \in X.\mathit{ltcat}.\mathit{pcat}.\mathit{arr}\ A\ B) \in \mathit{Prop} \qquad \left(\begin{array}{l} X \in \mathit{LTAllegory} \\ A \in X.\mathit{ltcat}.\mathit{pcat}.\mathit{obj} \\ B \in X.\mathit{ltcat}.\mathit{pcat}.\mathit{obj} \end{array} \right)
$$

$$
\mathit{entireArr}\ R \equiv \begin{array}{l} \textbf{open}\ X\ \textbf{use}\ \mathit{ltcat}, \circ \\ \textbf{in}\ \begin{array}{l} \textbf{open}\ \mathit{ltcat}\ \textbf{use}\ \mathit{pcat} \\ \textbf{in}\ \begin{array}{l} \textbf{open}\ \mathit{pcat}\ \textbf{use}\ \subseteq, I, \bullet \\ \textbf{in}\ I \subseteq (R \bullet (R^{\circ})) \end{array} \end{array} \end{array}
$$

4.7.3 Functions

A *function* is a relation that is both simple and entire. Hence, it behaves as a total function in the common sense of the word.

Definition

$$
\mathit{functionArr}\ (R \in X.\mathit{ltcat}.\mathit{pcat}.\mathit{arr}\ A\ B) \in \mathit{Prop} \qquad \left(\begin{array}{l} X \in \mathit{LTAllegory} \\ A \in X.\mathit{ltcat}.\mathit{pcat}.\mathit{obj} \\ B \in X.\mathit{ltcat}.\mathit{pcat}.\mathit{obj} \end{array} \right)
$$

$$
\mathit{functionArr}\ R \equiv \mathit{simpleArr}\ R\ \&\ \mathit{entireArr}\ R
$$

4.8 Tabular LT-allegories

4.8.1 Tabulations

A *tabulation* of a relation is a pair of functions that allows one to think of the arrow as a binary predicate. You can recover, in an abstract sense, each pair of individual arguments over which the relation holds by means of the two tabulation functions. Tabulations are then used to do pointwise proofs in what is a point-free calculus of relations.

Definition

$$tabulationOf\,(R \in X.ltcat.pcat.arr\,A\,B) \in Type \qquad \left(\begin{array}{l} X \in LTAllegory \\ A \in X.ltcat.pcat.obj \\ B \in X.ltcat.pcat.obj \end{array}\right)$$

$$tabulationOf\,R \equiv \quad \begin{array}{l} \textbf{sig} \\ \quad \textbf{open}\ X\ \textbf{use}\ ltcat, \cap, \circ \\ \quad \textbf{open}\ ltcat\ \textbf{use}\ pcat \\ \quad \textbf{open}\ pcat\ \textbf{use}\ obj, arr, \subseteq, I, \bullet \\ \quad C \in obj \\ \quad f \in arr\,C\,A \\ \quad g \in arr\,C\,B \\ \quad fIsFun \in functionArr\,f \\ \quad gIsFun \in functionArr\,g \\ \quad arrDecomp \in R = ((f^\circ) \bullet g) \\ \quad tabsId \in ((f \bullet (f^\circ)) \cap (g \bullet (g^\circ))) = I \end{array}$$

4.8.2 Tabular LT-allegories

A *tabular LT-allegory* is an LT-allegory in which every arrow has a tabulation.

Definition

$$TabularAllegory \in \#2$$

$$\begin{array}{l} TabularAllegory \equiv \\ \quad \textbf{sig} \\ \quad\quad tabal \in LTAllegory \\ \quad\quad allTabuls \in (A, B \in tabal.ltcat.pcat.obj, R \in tabal.ltcat.pcat.arr\,A\,B) \to \\ \quad\quad\quad tabulationOf\,R \end{array}$$

4.9 Unitary LT-allegories

A *unit* in an LT-allegory is a special object such that the biggest homogeneous relation over it is the identity. Also, for every other object there must exist an entire arrow between that object and the unit. A unit behaves as an abstract singleton set. An LT-allegory that has a unit is called *unitary*.

Definition

$UnitaryAllegory \in \#2$

$$UnitaryAllegory \equiv \begin{array}{l} \textbf{sig} \\ \quad algy \in LTAllegory \\ \quad \textbf{open}\ algy\ \textbf{use}\ ltcat \\ \quad \textbf{open}\ ltcat\ \textbf{use}\ pcat \\ \quad \textbf{open}\ pcat\ \textbf{use}\ obj, arr, \subseteq, I \\ \quad unit \in obj \\ \quad unitIsGreat \in (f \in arr\ unit\ unit) \to f \subseteq I \\ \quad unitHasArrow \in (A \in obj) \to arr\ A\ unit \\ \quad arrIsEntire \in (A \in obj) \to entireArr\ (unitHasArrow\ A) \end{array}$$

4.10 The purpose of units and tabulations

We can now explain how unitary tabular allegories are actually needed to ensure the models of the abstract notion of allegory are the intended ones (relations on sets). A Horn-sentence in an allegory is a formula with the shape:

$$E_1 = D_1 \wedge E_2 = D_2 \wedge \ldots \wedge E_n = D_n \Rightarrow E_{n+1} = D_{n+1}$$

where E_i and D_i are allegory formulas formed by composition, the allegory operators, tabulations and units. There is a meta-theorem (see [31]) that says that a Horn-sentence holds for every unitary tabular allegory if and only if it is true for the specific allegory of relations over (classical) sets. Attempting a similar proof for the constructive case is beyond the scope of this work due to its complexity, but we hope to explore the issue in future work.

4.11 Division LT-allegories

Explanation

The *left residual* of two arrows is defined as the biggest solution to an inequation on a LT-allegory of the form $X \cdot S \subseteq R$, where X is the unknown. A LT-allegory provided with a left residual operation is called a *division allegory*.

Definition

DivisionAllegory $\in$ #2

DivisionAllegory $\equiv$
sig
$\quad$ *distalgy* $\in$ *DistribAllegory*
$\quad$ **open** *distalgy* **use** *algy*
$\quad$ **open** *algy* **use** *ltcat*
$\quad$ **open** *ltcat* **use** *pcat*
$\quad$ **open** *pcat* **use** *obj*, *arr*, $\subseteq$, $\bullet$
$\quad$ $\mathit{leftRes} \in (A, B, C \in \mathit{obj}) \to \mathit{arr}\,A\,C \to \mathit{arr}\,B\,C \to \mathit{arr}\,A\,B$
$\quad$ $\mathit{defLeftRes} \in (A, B, C \in \mathit{obj}, R \in \mathit{arr}\,A\,C, S \in \mathit{arr}\,B\,C, X \in \mathit{arr}\,A\,B) \to$
$\qquad$ $((X \bullet S) \subseteq R) \leftrightarrow (X \subseteq \mathit{leftRes}\,A\,B\,C\,R\,S)$

4.12 Power LT-allegories

A *power LT-allegory* is an LT-allegory in which it is possible to deal with subsets and power sets in an abstract sense. It has, for every object A, a corresponding *power object* $P\,A$ (*powObject* in the definition). It is provided with a *membership relation* $\in$ too (*memberRel* in the definition). For each relation, there is a corresponding *power relation* (such that an element is intuitively related, via the power relation, to the set of all elements that are related to it via the original relation, and this property is captured by the relational axiom *univPptyPower* below). The power relation is required to be a function, besides (the proof object for this is *isFunPowTransp*).

Definition

PowerAllegory $\in$ #2

PowerAllegory $\equiv$
sig
$\quad$ *algy* $\in$ *LTAllegory*
$\quad$ **open** *algy* **use** *ltcat*
$\quad$ **open** *ltcat* **use** *pcat*
$\quad$ **open** *pcat* **use** *obj*, *arr*, $\bullet$
$\quad$ $\mathit{powObject} \in \mathit{obj} \to \mathit{obj}$
$\quad$ $\mathit{powTranspose} \in (A, B \in \mathit{obj}) \to \mathit{arr}\,A\,B \to \mathit{arr}\,A\,(\mathit{powObject}\,B)$
$\quad$ $\mathit{isFunPowTransp} \in (A, B \in \mathit{obj}, R \in \mathit{arr}\,A\,B) \to$
$\qquad$ $\mathit{functionArr}\,(\mathit{powTranspose}\,A\,B\,R)$
$\quad$ $\mathit{memberRel} \in (A \in \mathit{obj}) \to \mathit{arr}\,(\mathit{powObject}\,A)\,A$
$\quad$ $\mathit{univPptyPower} \in (A, B \in \mathit{obj}, R \in \mathit{arr}\,A\,B, f \in \mathit{arr}\,A\,(\mathit{powObject}\,B)) \to$
$\qquad$ $(f = \mathit{powTranspose}\,A\,B\,R) \leftrightarrow ((f \bullet \mathit{memberRel}\,B) = R)$

4.13 Boolean LT-allegories

Explanation

A *Boolean LT-allegory* is a LT-allegory with a *negation* operation (*neg* below). This operation corresponds to a classical complement operation. It is required to be order-reversing, an involution, and to satisfy a de Morgan law. These properties are captured by the three relational axioms that form the rest of the definition.

Definition

$\mathit{BooleanAllegory} \in \#2$

$$\mathit{BooleanAllegory} \equiv \begin{array}{l} \textbf{sig} \\ \quad \mathit{distalgy} \in \mathit{DistribAllegory} \\ \quad \textbf{open}\ \mathit{distalgy}\ \textbf{use}\ \mathit{algy}, \cup \\ \quad \textbf{open}\ \mathit{algy}\ \textbf{use}\ \mathit{ltcat}, \cap \\ \quad \textbf{open}\ \mathit{ltcat}\ \textbf{use}\ \mathit{pcat} \\ \quad \textbf{open}\ \mathit{pcat}\ \textbf{use}\ \mathit{obj}, \mathit{arr}, \subseteq \\ \quad \mathit{neg} \in (A, B \in \mathit{obj}) \to \mathit{arr}\, A\, B \to \mathit{arr}\, A\, B \\ \quad \mathit{isOrdRevNeg} \in (A, B \in \mathit{obj}, R, S \in \mathit{arr}\, A\, B) \to \\ \qquad (R \subseteq S) \to \mathit{neg}\, A\, B\, S \subseteq \mathit{neg}\, A\, B\, R \\ \quad \mathit{deMorgan} \in (A, B \in \mathit{obj}, R, S \in \mathit{arr}\, A\, B) \to \\ \qquad \mathit{neg}\, A\, B\, (R \cup S) = (\mathit{neg}\, A\, B\, R \cap \mathit{neg}\, A\, B\, S) \\ \quad \mathit{isInvolNeg} \in (A, B \in \mathit{obj}, R \in \mathit{arr}\, A\, B) \to \\ \qquad \mathit{neg}\, A\, B\, (\mathit{neg}\, A\, B\, R) = R \end{array}$$

4.14 LT-allegories of E-relations

In classical logic, the standard (and intended) allegory is the allegory of sets and relations. In this chapter we formally present the constructive analogue of this, the LT-allegory of setoids and E-relations. We also explore some properties of it, and show that we have the constructive analogue of a distributive, tabular, unitary allegory. We end this chapter by noting that the LT-allegory of E-relations is not a Boolean allegory and not a power allegory. All the particular E-relations used here are defined, with a few exceptions defined in this same chapter, in the previous section on E-relations.

The proof that E-relations form an LT-allegory is presented in successive sections. First in we build the precategory structure, that is, we define the type of objects as the type of setoids, the setoid of arrows as the set of equivalence preserving relations, identity as the identity relation, and composition as composition of relations. We then define inclusion of relations as the preorder of the LT-allegory structure, and equality of relations as double inclusion, also proving that in that way we get an equivalence relation. Then we prove the LT-category laws for E-relations, and finally we put all this together using signatures and structures. An important remark is that (as shown by the typings there) setoids and E-relations form a large LT-category.

After this, we consider both converse and intersection, along with their properties. We finally put all of this together to show that setoids and E-relations form an LT-allegory. Then we consider union of relations and the empty relation, along with their properties. With them, we can put things together to show that setoids and E-relations form a distributive allegory.

Following that, we deal with tabulations and how to move between a relation seen as a binary predicate and a relation seen as a set of pairs. The E-relations for both directions are defined, and then we prove that they satisfy the definition for a tabular LT-allegory. Then we deal with another kind of LT-allegory, the unitary ones, and we show they can readily be constructed by using the singleton setoid. As a final positive result, we establish the basic property of relational left residuals, which is used to show that setoids and E-relations form a division LT-allegory.

4.14.1 Precategories of E-relations

Setoids and E-relations over them form a precategory. The objects of the precategory are setoids, the arrows E-relations over them, the inequality is E-relation inclusion, identity arrows are the identity E-relations, and arrow composition is E-relation composition.

It is worth mentioning that setoids and E-relations over them also form an E-category. In fact, if one were to define all the several kinds of E-allegories instead of working with the LT-allegories, all that will be said in this section about setoids and E-relations being one such kind of LT-allegory also holds for the corresponding E-allegory.

$SetoidPrecat \in LargePrecat$

$$SetoidPrecat \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} obj \equiv Setoid \\ arr \equiv ERelation \\ \subseteq \equiv \subseteq \\ I \equiv I_= \\ \bullet \equiv \bullet_= \end{array} \right. \end{array}$$

In the preceding definition, the last three fields denote the instantiation in the signature for large precategories of the preorder, identity, and composition fields with their E-relational values.

4.14.2 Inclusion of relations is a preorder

Here we will use reflexivity and transitivity of implication, which are provided by the Alfa library as *reflImp* and *tranImp* respectively.

Reflexivity

Proposition Inclusion of relations is reflexive.

Proof

$$\mathit{isReflLTRel} \in R \subseteq R \qquad \begin{pmatrix} A \in \mathit{Set} \\ B \in \mathit{Set} \\ R \in \mathit{HetRel}\ A\ B \end{pmatrix}$$

$$\mathit{isReflLTRel} \equiv \lambda\, a\, b \rightarrow \mathit{reflImp}\ (R\ a\ b)$$

Transitivity

Proposition Inclusion of relations is transitive.

Proof

$$\mathit{isTransLTRel} \in (R \subseteq S) \rightarrow (S \subseteq T) \rightarrow R \subseteq T \qquad \begin{pmatrix} A \in \mathit{Set} \\ B \in \mathit{Set} \\ R \in \mathit{HetRel}\ A\ B \\ S \in \mathit{HetRel}\ A\ B \\ T \in \mathit{HetRel}\ A\ B \end{pmatrix}$$

$$\mathit{isTransLTRel} \equiv \lambda\, \mathit{hRS}\ \mathit{hST}\ a\ b \rightarrow \mathit{tranImp}\ (R\ a\ b)\ (S\ a\ b)\ (a\ T\ b)\ (\mathit{hRS}\ a\ b)\ (\mathit{hST}\ a\ b)$$

4.14.3 Equality of relations is an equivalence

We should remark here that our equality is extensional equality of relations: two relations are equal if and only if they hold for the same pairs of elements.

Reflexivity

Proposition Equality of relations is reflexive.

Proof

$$isReflEqRel \in R = R \quad \begin{pmatrix} A \in Set \\ B \in Set \\ R \in HetRel\ A\ B \end{pmatrix}$$

$$isReflEqRel \equiv \quad \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv isReflLTRel \\ snd \equiv isReflLTRel \end{array}\right.\end{array}$$

Symmetry

Proposition Equality of relations is symmetric.

Proof

$$isSymmEqRel \in (R = S) \to S = R \quad \begin{pmatrix} A \in Set \\ B \in Set \\ R \in HetRel\ A\ B \\ S \in HetRel\ A\ B \end{pmatrix}$$

$$isSymmEqRel \equiv \lambda\, h \to \quad \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv h.snd \\ snd \equiv h.fst \end{array}\right.\end{array}$$

Transitivity

Proposition Equality of relations is transitive.

Proof

$$isTransEqRel \in (R = S) \to (S = T) \to R = T \quad \begin{pmatrix} A \in Set \\ B \in Set \\ R \in HetRel\ A\ B \\ S \in HetRel\ A\ B \\ T \in HetRel\ A\ B \end{pmatrix}$$

$$isTransEqRel \equiv \lambda\, hRS\ hST \to \quad \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv isTransLTRel\ hRS.fst\ hST.fst \\ snd \equiv isTransLTRel\ hST.snd\ hRS.snd \end{array}\right.\end{array}$$

4.14.4 Properties of composition

Associativity

Proposition Composition of relations is associative.

Proof

$$\mathit{isAssocComposRel} \in ((R \bullet S) \bullet T) = (R \bullet (S \bullet T)) \qquad \left(\begin{array}{l} A \in \mathit{Set} \\ B \in \mathit{Set} \\ C \in \mathit{Set} \\ D \in \mathit{Set} \\ R \in \mathit{HetRel}\,A\,B \\ S \in \mathit{HetRel}\,B\,C \\ T \in \mathit{HetRel}\,C\,D \end{array}\right)$$

$\mathit{isAssocComposRel} \equiv$

$$\begin{array}{l}
\textbf{struct} \\
\left[\begin{array}{l}
\mathit{fst} \equiv \lambda\, a\, d\, h \;\rightarrow \\
\qquad \textbf{struct} \\
\qquad \left[\ \mathit{fst} \equiv h.\mathit{snd}.\mathit{fst}.\mathit{fst} \right. \\
\qquad \left[\ \mathit{snd} \equiv \begin{array}{l}
\textbf{struct} \\
\left[\ \mathit{fst} \equiv h.\mathit{snd}.\mathit{fst}.\mathit{snd}.\mathit{fst} \right. \\
\left[\ \mathit{snd} \equiv \begin{array}{l}
\textbf{struct} \\
\left[\ \mathit{fst} \equiv h.\mathit{fst} \right. \\
\left[\ \mathit{snd} \equiv \begin{array}{l}
\textbf{struct} \\
\left[\begin{array}{l} \mathit{fst} \equiv h.\mathit{snd}.\mathit{fst}.\mathit{snd}.\mathit{snd} \\ \mathit{snd} \equiv h.\mathit{snd}.\mathit{snd} \end{array}\right.
\end{array}\right.
\end{array}\right.
\end{array}\right.
\end{array}\right. \\
\left[\begin{array}{l}
\mathit{snd} \equiv \lambda\, a\, d\, h \;\rightarrow \\
\qquad \textbf{struct} \\
\qquad \left[\ \mathit{fst} \equiv h.\mathit{snd}.\mathit{snd}.\mathit{fst} \right. \\
\qquad \left[\ \mathit{snd} \equiv \begin{array}{l}
\textbf{struct} \\
\left[\ \mathit{fst} \equiv \begin{array}{l}
\textbf{struct} \\
\left[\ \mathit{fst} \equiv h.\mathit{fst} \right. \\
\left[\ \mathit{snd} \equiv \begin{array}{l}
\textbf{struct} \\
\left[\begin{array}{l} \mathit{fst} \equiv h.\mathit{snd}.\mathit{fst} \\ \mathit{snd} \equiv h.\mathit{snd}.\mathit{snd}.\mathit{snd}.\mathit{fst} \end{array}\right.
\end{array}\right.
\end{array}\right. \\
\left[\ \mathit{snd} \equiv h.\mathit{snd}.\mathit{snd}.\mathit{snd}.\mathit{snd} \right.
\end{array}\right.
\end{array}\right.
\end{array}$$

The proof is actually simpler than it looks: it consists of rearranging the component subproofs and bridging elements for the nested composition on one side of the equality to get the proof object for the other side.

Monotonicity

Proposition Composition of relations is monotone with respect to relation inclusion.

Proof

$$isMonotComposRel \in (R1 \subseteq R2) \to (S1 \subseteq S2) \to (R1 \bullet S1) \subseteq (R2 \bullet S2)$$
$$\left(\begin{array}{l} A \in Set \\ B \in Set \\ C \in Set \\ R1 \in HetRel\,A\,B \\ R2 \in HetRel\,A\,B \\ S1 \in HetRel\,B\,C \\ S2 \in HetRel\,B\,C \end{array}\right)$$

$$isMonotComposRel \equiv \lambda\, hR\, hS\, a\, c\, h \to \mathbf{struct}\left[\begin{array}{l} fst \equiv h.fst \\ snd \equiv \mathbf{struct}\left[\begin{array}{l} fst \equiv hR\, a\, h.fst\, h.snd.fst \\ snd \equiv hS\, h.fst\, c\, h.snd.snd \end{array}\right] \end{array}\right]$$

Left identity

Proposition The identity relation is a left identity for relation composition.

Proof

$$isLeftId \in (I = \bullet = R) = R \qquad \left(\begin{array}{l} A \in Setoid \\ B \in Setoid \\ R \in ERelation\,A\,B \end{array}\right)$$

$$isLeftId \equiv \mathbf{struct}\left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h \to R.resp\, h.fst\, a\, b\, b\; (A.sym\, a\, h.fst\, h.snd.fst)\; (B.ref\, b)\; h.snd.snd \\ snd \equiv \lambda\, a\, b\, h \to \mathbf{struct}\left[\begin{array}{l} fst \equiv a \\ snd \equiv \mathbf{struct}\left[\begin{array}{l} fst \equiv A.ref\, a \\ snd \equiv h \end{array}\right] \end{array}\right] \end{array}\right]$$

We remark that in the above $\bullet=$ is E-relation composition, as defined in the previous chapter.

Right identity

Proposition The identity relation is a right identity for relation composition.

Proof

$$isRightId \in (R \bullet = I{=}) = R \quad \begin{pmatrix} A \in Setoid \\ B \in Setoid \\ R \in ERelation\ A\ B \end{pmatrix}$$

$$isRightId \equiv \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h\ \rightarrow\ R.resp\ a\ a\ h.fst\ b\ (A.ref\ a)\ h.snd.snd\ h.snd.fst \\ snd \equiv \lambda\, a\, b\, h\ \rightarrow\ \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv b \\ snd \equiv \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv h \\ snd \equiv B.ref\ b \end{array}\right.\end{array} \end{array}\right.\end{array} \end{array}\right.\end{array}$$

4.14.5 LT-categories of E-relations

Proposition

The precategory of setoids and their E-relations forms an LT-category.

Proof

$$SetoidLTCat \in LargeLTCat$$

$$SetoidLTCat \equiv \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} pcat \equiv SetoidPrecat \\ ltIsRef \equiv \lambda\, A\, B\, f\ \rightarrow\ isReflLTRel \\ ltIsTra \equiv \lambda\, A\, B\, f\, g\, h\ \rightarrow\ isTransLTRel \\ compIsMonot \equiv \lambda\, A\, B\, C\, f1\, f2\, g1\, g2\ \rightarrow\ isMonotComposRel \\ isLId \equiv isLeftId \\ isRId \equiv isRightId \\ isAso \equiv \lambda\, A\, B\, C\, D\, f\, g\, h\ \rightarrow\ isAssocComposRel \end{array}\right.\end{array}$$

4.14.6 Properties of converse

Involution

Proposition Converse of relations is an involutive operation.

Proof

$$isInvolConvRel \in ((R^\circ)^\circ) = R \quad \begin{pmatrix} A \in Set \\ B \in Set \\ R \in HetRel\, A\, B \end{pmatrix}$$

$$isInvolConvRel \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h \;\rightarrow\; h \\ snd \equiv \lambda\, a\, b\, h \;\rightarrow\; h \end{array} \right. \end{array}$$

In the above proof, both directions of the doble inclusion are immediate, since swapping the arguments of R twice in a row is automatically reduced by Alfa to the original situation.

Order preservation

Proposition Converse of relations is order-preserving with respect to relation inclusion.

Proof

$$isOrdPreservConvRel \in (R \subseteq S) \leftrightarrow ((R^\circ) \subseteq (S^\circ)) \quad \begin{pmatrix} A \in Set \\ B \in Set \\ R \in HetRel\, A\, B \\ S \in HetRel\, A\, B \end{pmatrix}$$

$$isOrdPreservConvRel \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv \lambda\, h\, b\, a\, h' \;\rightarrow\; h\, a\, b\, h' \\ snd \equiv \lambda\, h\, a\, b\, h' \;\rightarrow\; h\, b\, a\, h' \end{array} \right. \end{array}$$

Contravariance

Proposition Converse of relations is contravariant with respect to relation composition.

Proof

$$isContravarConvRel \in ((R \bullet S)^{\circ}) = ((S^{\circ}) \bullet (R^{\circ})) \qquad \begin{pmatrix} A \in Set \\ B \in Set \\ C \in Set \\ R \in HetRel\,A\,B \\ S \in HetRel\,B\,C \end{pmatrix}$$

$$isContravarConvRel \equiv \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv \lambda\, c\, a\, h \rightarrow \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv h.fst \\ snd \equiv \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv h.snd.snd \\ snd \equiv h.snd.fst \end{array}\right.\end{array}\end{array}\right.\end{array} \\ snd \equiv \lambda\, c\, a\, h \rightarrow \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv h.fst \\ snd \equiv \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv h.snd.snd \\ snd \equiv h.snd.fst \end{array}\right.\end{array}\end{array}\right.\end{array} \end{array}\right.\end{array}$$

This is another proof in which the basic idea is to rearrange the pieces of one proof object for composition in the shape of a proof object for another composition.

4.14.7 Properties of intersection

Greatest lower bound

Proposition Intersection of relations is the greatest lower bound of its arguments, with respect to relation inclusion.

Proof

$$isGLBIntersec \in (X \subseteq (R \cap S)) \leftrightarrow ((X \subseteq R) \;\&\; (X \subseteq S)) \qquad \begin{pmatrix} A \in Set \\ B \in Set \\ R \in HetRel\,A\,B \\ S \in HetRel\,A\,B \\ X \in HetRel\,A\,B \end{pmatrix}$$

$$
isGLBIntersec \equiv \begin{array}{l}\textbf{struct}\\ \left[\; fst \equiv \lambda\, h \;\rightarrow\; \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h' \;\rightarrow\; (h\, a\, b\, h').fst \\ snd \equiv \lambda\, a\, b\, h' \;\rightarrow\; (h\, a\, b\, h').snd \end{array}\right.\end{array}\right. \\ \left[\; snd \equiv \lambda\, h\, a\, b\, h' \;\rightarrow\; \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv h.fst\, a\, b\, h' \\ snd \equiv h.snd\, a\, b\, h' \end{array}\right.\end{array}\right.\end{array}
$$

Modular law

Proposition The modular law of LT-allegories holds for relations and their operations.

Proof

$$
modularLaw \;\in ((R \,\bullet\, S) \cap T) \subseteq ((R \cap (T \,\bullet\, (S^{\circ}))) \,\bullet\, S) \qquad \left(\begin{array}{l} A \in Set \\ B \in Set \\ C \in Set \\ R \in HetRel\, A\, B \\ S \in HetRel\, B\, C \\ T \in HetRel\, A\, C \end{array}\right)
$$

$$
\begin{array}{l} modularLaw \equiv \lambda\, a\, c\, h \;\rightarrow \\ \textbf{struct} \\ \left[\; fst \equiv h.fst.fst \right. \\ \left[\; snd \equiv \begin{array}{l}\textbf{struct}\\ \left[\; fst \equiv \begin{array}{l}\textbf{struct}\\ \left[\; fst \equiv h.fst.snd.fst \right. \\ \left[\; snd \equiv \begin{array}{l}\textbf{struct}\\ \left[\; fst \equiv c \right. \\ \left[\; snd \equiv \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv h.snd \\ snd \equiv h.fst.snd.snd \end{array}\right.\end{array}\right.\end{array}\right.\end{array}\right. \\ \left[\; snd \equiv h.fst.snd.snd \right.\end{array}\right.\end{array}
$$

This is yet another example of a proof consisting of rearraging already existing pieces of a proof object into the shape of a new proof object. Here we have, besides the parts of a composition proof object, those of a conjunction one.

4.14.8 LT-allegories of E-relations

Proposition

The LT-category of setoids and their E-relations forms an LT-allegory, with relation intersection as meet, and relation converse as converse.

Proof

$SetoidAllegory \in LTAllegory$

$$SetoidAllegory \equiv \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} ltcat \equiv SetoidLTCat \\ \cap \equiv \cap = \\ \circ \equiv \circ = \\ defMeet \equiv \lambda\, A\, B\, f\, g\, x \;\rightarrow\; isGLBIntersec \\ isInvolConv \equiv \lambda\, A\, B\, f \;\rightarrow\; isInvolConvRel \\ isOrdPresConv \equiv \lambda\, A\, B\, f\, g \;\rightarrow\; isOrdPreservConvRel \\ isContravConv \equiv \lambda\, A\, B\, C\, f\, g \;\rightarrow\; isContravarConvRel \\ modLaw \equiv \lambda\, A\, B\, C\, f\, g\, h \;\rightarrow\; modularLaw \end{array}\right.\end{array}$$

4.14.9 Properties of union

Least upper bound

Proposition The union of relations is the least upper bound of its arguments, with respect to relation inclusion.

Proof

$$isLUBUnion \in ((R \cup S) \subseteq X) \leftrightarrow ((R \subseteq X) \;\&\; (S \subseteq X)) \qquad \left(\begin{array}{l} A \in Set \\ B \in Set \\ R \in HetRel\, A\, B \\ S \in HetRel\, A\, B \\ X \in HetRel\, A\, B \end{array}\right)$$

$$isLUBUnion \equiv \begin{array}{l}\mathbf{struct}\\ \left[\begin{array}{l} fst \equiv \lambda\, h \;\rightarrow\; \begin{array}{l}\mathbf{struct}\\ \left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h' \;\rightarrow\; h\, a\, b\; (\mathbf{inl}\, h')\\ snd \equiv \lambda\, a\, b\, h' \;\rightarrow\; h\, a\, b\; (\mathbf{inr}\, h')\end{array}\right.\end{array}\\ snd \equiv \lambda\, h\, a\, b\, h' \;\rightarrow\; \begin{array}{l}\mathbf{case}\; h'\;\mathbf{of}\\ \quad \mathbf{inl}\, byR \;\rightarrow\; h.fst\, a\, b\, byR\\ \quad \mathbf{inr}\, byS \;\rightarrow\; h.snd\, a\, b\, byS\end{array}\end{array}\right.\end{array}$$

Distributivity of intersection

Proposition Intersection of relations is distributive over union on the left.

Proof

$$isDistrIntersUnion \;\in (R \cap (S \cup T)) = ((R \cap S) \cup (R \cap T)) \quad \left(\begin{array}{l} A \in Set\\ B \in Set\\ R \in HetRel\, A\, B\\ S \in HetRel\, A\, B\\ T \in HetRel\, A\, B\end{array}\right)$$

$$\begin{array}{l} isDistrIntersUnion \equiv\\ \quad \mathbf{struct}\\ \quad \left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h \;\rightarrow\; \begin{array}{l}\mathbf{case}\; h.snd\;\mathbf{of}\\ \quad \mathbf{inl}\, byS \;\rightarrow\; \mathbf{inl}\left(\begin{array}{l}\mathbf{struct}\\ \left[\begin{array}{l} fst \equiv h.fst\\ snd \equiv byS\end{array}\right.\end{array}\right)\\ \quad \mathbf{inr}\, byT \;\rightarrow\; \mathbf{inr}\left(\begin{array}{l}\mathbf{struct}\\ \left[\begin{array}{l} fst \equiv h.fst\\ snd \equiv byT\end{array}\right.\end{array}\right)\end{array}\\ snd \equiv \lambda\, a\, b\, h \;\rightarrow\; \begin{array}{l}\mathbf{case}\; h\;\mathbf{of}\\ \quad \mathbf{inl}\, byRS \;\rightarrow\; \begin{array}{l}\mathbf{struct}\\ \left[\begin{array}{l} fst \equiv byRS.fst\\ snd \equiv \mathbf{inl}\, byRS.snd\end{array}\right.\end{array}\\ \quad \mathbf{inr}\, byRT \;\rightarrow\; \begin{array}{l}\mathbf{struct}\\ \left[\begin{array}{l} fst \equiv byRT.fst\\ snd \equiv \mathbf{inr}\, byRT.snd\end{array}\right.\end{array}\end{array}\end{array}\right.\end{array}$$

Distributivity of composition

Proposition Composition of relations distributes over union on the left.

Proof

$$isDistrComposUnion \in (R \bullet (S \cup T)) = ((R \bullet S) \cup (R \bullet T)) \qquad \left(\begin{array}{l} A \in Set \\ B \in Set \\ C \in Set \\ R \in HetRel\, A\, B \\ S \in HetRel\, B\, C \\ T \in HetRel\, B\, C \end{array}\right)$$

$isDistrComposUnion \equiv$

$$\begin{array}{l}
\textbf{struct} \\
\left[\begin{array}{l}
fst \equiv \lambda\, a\, c\, h \rightarrow \begin{array}{l}
\textbf{case}\ h.snd.snd\ \textbf{of} \\
\begin{array}{ll}
\textbf{inl}\ byS \rightarrow \textbf{inl} & \left(\begin{array}{l}
\textbf{struct} \\
\left[fst \equiv h.fst \right. \\
\left[snd \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv h.snd.fst \\ snd \equiv byS \end{array}\right. \end{array} \right.
\end{array}\right) \\
\textbf{inr}\ byT \rightarrow \textbf{inr} & \left(\begin{array}{l}
\textbf{struct} \\
\left[fst \equiv h.fst \right. \\
\left[snd \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv h.snd.fst \\ snd \equiv byT \end{array}\right. \end{array} \right.
\end{array}\right)
\end{array}
\end{array}
\end{array}\right. \\
\left[\begin{array}{l}
snd \equiv \lambda\, a\, c\, h \rightarrow \begin{array}{l}
\textbf{case}\ h\ \textbf{of} \\
\begin{array}{ll}
\textbf{inl}\ byRS \rightarrow & \begin{array}{l}
\textbf{struct} \\
\left[fst \equiv byRS.fst \right. \\
\left[snd \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv byRS.snd.fst \\ snd \equiv \textbf{inl}\ byRS.snd.snd \end{array}\right. \end{array} \right.
\end{array} \\
\textbf{inr}\ byRT \rightarrow & \begin{array}{l}
\textbf{struct} \\
\left[fst \equiv byRT.fst \right. \\
\left[snd \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv byRT.snd.fst \\ snd \equiv \textbf{inr}\ byRT.snd.snd \end{array}\right. \end{array} \right.
\end{array}
\end{array}
\end{array}
\end{array}\right.
\end{array}$$

4.14.10 Properties of the empty relation

Zero for intersection

Proposition The empty relation is a right zero for relation intersection.

Proof

$$isZeroInters \in (R \cap \emptyset) = \emptyset \qquad \begin{pmatrix} A \in Set \\ B \in Set \\ R \in HetRel\,A\,B \end{pmatrix}$$

$$isZeroInters \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h \to h.snd \\ snd \equiv \lambda\, a\, b\, h \to elimAbsurd\ (\cap\, R\, \emptyset\, a\, b)\ h \end{array} \right. \end{array}$$

Zero for union

Proposition The empty relation is a right zero for relation union.

Proof

$$isZeroUnion \in (R \cup \emptyset) = R \qquad \begin{pmatrix} A \in Set \\ B \in Set \\ R \in HetRel\,A\,B \end{pmatrix}$$

$$isZeroUnion \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h \to \begin{array}{l} \textbf{case}\ h\ \textbf{of} \\ \quad \textbf{inl}\ byR \to byR \\ \quad \textbf{inr}\ byE \to elimAbsurd\ (R\, a\, b)\ byE \end{array} \\ snd \equiv \lambda\, a\, b\, h \to \textbf{inl}\ h \end{array} \right. \end{array}$$

Zero for composition

Proposition The empty relation is a right zero for relation composition.

Proof

$$isZeroComposRel \in (R \bullet \emptyset) = \emptyset \qquad \begin{pmatrix} A \in Set \\ B \in Set \\ C \in Set \\ R \in HetRel\,A\,B \end{pmatrix}$$

$$isZeroComposRel \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h \to h.snd.snd \\ snd \equiv \lambda\, a\, b\, h \to elimAbsurd\ (\bullet\, R\, \emptyset\, a\, b)\ h \end{array} \right. \end{array}$$

4.14.11 Distributive LT-allegories of E-relations

Proposition

The LT-allegory of setoids and their E-relations forms a distributive LT-allegory, with relation union as join, and the empty relation as the zero arrow.

Proof

$\mathit{SetoidDistrAllegory} \ \in \mathit{DistribAllegory}$

$$\mathit{SetoidDistrAllegory} \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} \mathit{algy} \equiv \mathit{SetoidAllegory} \\ \cup \equiv \cup = \\ \emptyset \equiv \emptyset = \\ \mathit{defJoin} \equiv \lambda\, A\, B\, f\, g\, x \ \rightarrow \ \mathit{isLUBUnion} \\ \mathit{isDistrMeetJoin} \equiv \lambda\, A\, B\, f\, g\, h \ \rightarrow \ \mathit{isDistrIntersUnion} \\ \mathit{isDistrComposJoin} \equiv \lambda\, A\, B\, C\, f\, g\, h \ \rightarrow \ \mathit{isDistrComposUnion} \\ \mathit{isZeroMeet} \equiv \lambda\, A\, B\, f \ \rightarrow \ \mathit{isZeroInters} \\ \mathit{isZeroJoin} \equiv \lambda\, A\, B\, f \ \rightarrow \ \mathit{isZeroUnion} \\ \mathit{isZeroCompos} \equiv \lambda\, A\, B\, C\, f \ \rightarrow \ \mathit{isZeroComposRel} \end{array}\right. \end{array}$$

In the preceding definition, $\cup =$ and $\emptyset =$ are E-relation union and the empty E-relation, respectively.

4.14.12 Tabular LT-allegories of E-relations

Internalisation of a relation

Explanation The *internalisation* of an E-relation between two setoids is a new setoid, formed by the pair of elements of the first two setoids for which the E-relation holds. In set theory, the internalisation of R is the set $\{(a, b)|aRb\}$.

Definition

$$\mathit{internalise}\ (R\ \in\ \mathit{ERelation}\ A\ B) \in \mathit{Setoid} \qquad \left(\begin{array}{l} A \in \mathit{Setoid} \\ B \in \mathit{Setoid} \end{array}\right)$$

$\mathit{internalise}\ R \equiv$

struct

$$\left[\begin{array}{l} Elem \equiv \begin{array}{l}\textbf{sig}\\ \quad fst \in |A| \\ \quad snd \in |B| \\ \quad inRel \in R.rel\, fst\, snd\end{array}\end{array}\right.$$

$$\left[\; Equal \equiv \lambda\, x\, y \;\rightarrow == \; A\, x.fst\, y.fst \;\&\; == \; B\, x.snd\, y.snd\right.$$

$$\left[\; ref \equiv \lambda\, x \;\rightarrow \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv A.ref\, x.fst \\ snd \equiv B.ref\, x.snd\end{array}\right.\end{array}\right.$$

$$\left[\; sym \equiv \lambda\, x1\, x2\, h \;\rightarrow \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv A.sym\, x1.fst\, x2.fst\, h.fst \\ snd \equiv B.sym\, x1.snd\, x2.snd\, h.snd\end{array}\right.\end{array}\right.$$

$$\left[\begin{array}{l} tran \equiv \lambda\, x1\, x2\, x3\, h12\, h23 \;\rightarrow \\ \quad \textbf{struct} \\ \quad \left[\begin{array}{l} fst \equiv A.tran\, x1.fst\, x2.fst\, x3.fst\, h12.fst\, h23.fst \\ snd \equiv B.tran\, x1.snd\, x2.snd\, x3.snd\, h12.snd\, h23.snd\end{array}\right.\end{array}\right.$$

Projections from the internalisation

When we have a relation represented by its internalisation setoid, we can recover the elements that are related by the original relation by taking the pairs that are in the setoid and forming a relation between the projected out components of those pairs . We call these relational projections *uninternalisations* in the definitions.

Definition

$$uninternal1\ (R \in ERelation\, A\, B) \in ERelation\ (internalise\, R)\ A \qquad \left(\begin{array}{l} A \in Setoid \\ B \in Setoid\end{array}\right)$$

$$uninternal1\, R \equiv \begin{array}{l}\textbf{struct}\\ \left[\; rel \equiv \lambda\, x\, a \;\rightarrow == \; A\, x.fst\, a\right. \\ \left[\; resp \equiv \lambda\, x\, x'\, a1\, a2\, hX\, hA\, h \;\rightarrow \begin{array}{l} A.tran \\ \;\; x'.fst \\ \;\; x.fst \\ \;\; a2 \\ \;\; (A.sym\, x.fst\, x'.fst\, hX.fst) \\ \;\; (A.tran\, x.fst\, a1\, a2\, h\, hA)\end{array}\right.\end{array}$$

$$uninternal2\ (R \in ERelation\, A\, B) \in ERelation\ (internalise\, R)\ B \qquad \left(\begin{array}{l} A \in Setoid \\ B \in Setoid\end{array}\right)$$

$$
\mathit{uninternal2}\ R \equiv
\begin{array}{l}
\textbf{struct} \\
\left[\ \mathit{rel} \equiv \lambda\, x\, b\ \rightarrow == \ B\ x.\mathit{snd}\ b \right. \\
\left[\ \mathit{resp} \equiv \lambda\, x\, x'\, b1\, b2\, hX\, hB\, h\ \rightarrow
\begin{array}{l}
B.\mathit{tran} \\
\ \ x'.\mathit{snd} \\
\ \ x.\mathit{snd} \\
\ \ b2 \\
\ \ (B.\mathit{sym}\ x.\mathit{snd}\ x'.\mathit{snd}\ hX.\mathit{snd}) \\
\ \ (B.\mathit{tran}\ x.\mathit{snd}\ b1\ b2\ h\ hB)
\end{array}\right.
\end{array}
$$

Properties of the uninternalisations

Proposition The uninternalisation relations are functions in the relational sense.

Proof

$$
\mathit{isFunUninter1}\ \in\
\begin{array}{l}
(((\mathit{uninternal1}\ R^{\circ=})\ \bullet =\ \mathit{uninternal1}\ R)\ \subseteq\ I=)\ \&\\
(I=\ \subseteq\ (\mathit{uninternal1}\ R \bullet\ =\ (\mathit{uninternal1}\ R^{\circ=})))
\end{array}
$$

$$
\left(\begin{array}{l}
A \in \mathit{Setoid} \\
B \in \mathit{Setoid} \\
R \in \mathit{ERelation}\ A\ B
\end{array}\right)
$$

$$
\begin{array}{l}
\mathit{isFunUninter1} \equiv \\
\quad \textbf{struct} \\
\quad \left[\ \mathit{fst} \equiv \lambda\, a1\, a2\, h\ \rightarrow
\begin{array}{l}
A.\mathit{tran} \\
\ \ a1 \\
\ \ h.\mathit{fst}.\mathit{fst} \\
\ \ a2 \\
\ \ (A.\mathit{sym}\ h.\mathit{fst}.\mathit{fst}\ a1\ h.\mathit{snd}.\mathit{fst}) \\
\ \ h.\mathit{snd}.\mathit{snd}
\end{array}\right. \\
\quad \left[\ \mathit{snd} \equiv \lambda\, x\, x'\, h\ \rightarrow
\begin{array}{l}
\textbf{struct} \\
\left[\ \mathit{fst} \equiv x.\mathit{fst}\right. \\
\left[\ \mathit{snd} \equiv
\begin{array}{l}
\textbf{struct} \\
\left[\ \mathit{fst} \equiv A.\mathit{ref}\ x.\mathit{fst}\right. \\
\left[\ \mathit{snd} \equiv A.\mathit{sym}\ x.\mathit{fst}\ x'.\mathit{fst}\ h.\mathit{fst}\right.
\end{array}\right.
\end{array}\right.
\end{array}
$$

$$
\mathit{isFunUninter2}\ \in\
\begin{array}{l}
(((\mathit{uninternal2}\ R^{\circ=})\ \bullet =\ \mathit{uninternal2}\ R)\ \subseteq\ I=)\ \&\\
(I=\ \subseteq\ (\mathit{uninternal2}\ R \bullet\ =\ (\mathit{uninternal2}\ R^{\circ=})))
\end{array}
$$

$$
\left(\begin{array}{l}
A \in \mathit{Setoid} \\
B \in \mathit{Setoid} \\
R \in \mathit{ERelation}\ A\ B
\end{array}\right)
$$

$\mathit{isFunUninter2} \equiv$

$$\textbf{struct}\left[\begin{array}{ll} fst \equiv \lambda\, b1\, b2\, h \;\rightarrow & \begin{array}{l} B.tran \\ \;b1 \\ \;h.fst.snd \\ \;b2 \\ \;(B.sym\, h.fst.snd\, b1\, h.snd.fst) \\ \;h.snd.snd \end{array} \\ snd \equiv \lambda\, x\, x'\, h \;\rightarrow & \textbf{struct}\left[\begin{array}{l} fst \equiv x.snd \\ snd \equiv \textbf{struct}\left[\begin{array}{l} fst \equiv B.ref\, x.snd \\ snd \equiv B.sym\, x.snd\, x'.snd\, h.snd \end{array}\right] \end{array}\right] \end{array}\right]$$

Tabular LT-allegory of E-relations

Proposition The LT-allegory of setoids and their E-relations is a tabular allegory, with the internalisation setoid for a relation as the special object for the relation, and the uninternalisation projection E-relations as the tabulations of the relation.

Proof

$$relByTabuls\; \in R = ((uninternal1\, R^{\circ=}) \bullet = \; uninternal2\, R) \qquad \left(\begin{array}{l} A \in Setoid \\ B \in Setoid \\ R \in ERelation\, A\, B \end{array}\right)$$

$$relByTabuls \equiv \textbf{struct}\left[\begin{array}{ll} fst \equiv \lambda\, a\, b\, h \;\rightarrow & \textbf{struct}\left[\begin{array}{l} fst \equiv \textbf{struct}\left[\begin{array}{l} fst \equiv a \\ snd \equiv b \\ inRel \equiv h \end{array}\right] \\ snd \equiv \textbf{struct}\left[\begin{array}{l} fst \equiv A.ref\, a \\ snd \equiv B.ref\, b \end{array}\right] \end{array}\right] \\ snd \equiv \lambda\, a\, b\, h \;\rightarrow & R.resp\, h.fst.fst\, a\, h.fst.snd\, b\, h.snd.fst\, h.snd.snd\, h.fst.inRel \end{array}\right]$$

$$tabulsIdentity\; \in \left(\begin{array}{c} (uninternal1\, R \bullet = \; (uninternal1\, R^{\circ=})) \cap = \\ (uninternal2\, R \bullet = \; (uninternal2\, R^{\circ=})) \end{array}\right) = \quad (I{=})$$
$$\left(\begin{array}{l} A \in Setoid \\ B \in Setoid \\ R \in ERelation\, A\, B \end{array}\right)$$

$tabulsIdentity \equiv$

$$
\begin{array}{l}
\textbf{struct} \\
\left[\begin{array}{l}
fst \equiv \lambda\, x\, x'\, h \;\rightarrow\; \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv A.tran\; x.fst\; h.fst.fst\; x'.fst\; h.fst.snd.fst \\ \quad (A.sym\; x'.fst\; h.fst.fst\; h.fst.snd.snd) \\ snd \equiv B.tran\; x.snd\; h.snd.fst\; x'.snd\; h.snd.snd.fst \\ \quad (B.sym\; x'.snd\; h.snd.fst\; h.snd.snd.snd) \end{array}\right.\end{array} \\
snd \equiv \lambda\, x\, x'\, h \;\rightarrow\; \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l}
fst \equiv \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv x.fst \\ snd \equiv \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv A.ref\; x.fst \\ snd \equiv A.sym\; x.fst\; x'.fst\; h.fst \end{array}\right.\end{array}\end{array}\right.\end{array} \\
snd \equiv \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv x.snd \\ snd \equiv \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv B.ref\; x.snd \\ snd \equiv B.sym\; x.snd\; x'.snd\; h.snd \end{array}\right.\end{array}\end{array}\right.\end{array}
\end{array}\right.\end{array}
\end{array}\right.
\end{array}
$$

$SetoidTabulAllegory \;\in TabularAllegory$

$$
SetoidTabulAllegory \equiv \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l}
tabal \equiv SetoidAllegory \\
allTabuls \equiv \lambda\, A\, B\, R \;\rightarrow\; \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l}
C \equiv internalise\; R \\
f \equiv uninternal1\; R \\
g \equiv uninternal2\; R \\
fIsFun \equiv isFunUninter1 \\
gIsFun \equiv isFunUninter2 \\
arrDecomp \equiv relByTabuls \\
tabsId \equiv tabulsIdentity
\end{array}\right.\end{array}
\end{array}\right.\end{array}
$$

4.14.13 Unitary LT-allegories of E-relations

Proposition

The LT-allegory of setoids and E-relations is a unitary LT-allegory, with unit the singleton setoid, and the entire arrow going from every object to the unit equal to the constant function that always gives the element of the singleton.

Proof

$$SetoidUnitaryAllegory \in UnitaryAllegory$$

$$\begin{array}{l} SetoidUnitaryAllegory \equiv \\ \quad \textbf{struct} \\ \quad \left[\begin{array}{l} algy \equiv SetoidAllegory \\ unit \equiv \{!\} \\ unitIsGreat \equiv \lambda\, f\, x\, x'\, h \rightarrow \textbf{tt} \\ unitHasArrow \equiv \lambda\, A \rightarrow \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} rel \equiv \lambda\, a\, x \rightarrow T \\ resp \equiv \lambda\, a1\, a2\, x1\, x2\, hA\, hX\, h \rightarrow \textbf{tt} \end{array} \right. \end{array} \\ arrIsEntire \equiv \lambda\, A\, a1\, a2\, h \rightarrow \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv \textbf{tt} \\ snd \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv \textbf{tt} \\ snd \equiv \textbf{tt} \end{array} \right. \end{array} \end{array} \right. \end{array} \end{array} \right. \end{array}$$

In the preceding proof, {!} denotes the singleton setoid, which is defined in the standard Alfa library.

4.14.14 Properties of left residuals

Proposition

The left residual R/S is the biggest solution of the inequation $X \bullet S \subseteq R$.

Proof

$$isSolutIneqLeftRes \in ((X \bullet S) \subseteq R) \leftrightarrow (X \subseteq (R/S)) \qquad \begin{pmatrix} A \in Set \\ B \in Set \\ C \in Set \\ R \in HetRel\, A\, C \\ S \in HetRel\, B\, C \\ X \in HetRel\, A\, B \end{pmatrix}$$

$$isSolutIneqLeftRes \equiv$$

$$
\begin{array}{l}
\textbf{struct} \\
\left[\begin{array}{l}
\mathit{fst} \equiv \lambda\, h\, a\, b\, hX\, c\, hS \;\rightarrow\; h\, a\, c \left(\begin{array}{l}
\textbf{struct} \\
\left[\begin{array}{l}
\mathit{fst} \equiv b \\
\mathit{snd} \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} \mathit{fst} \equiv hX \\ \mathit{snd} \equiv hS \end{array}\right. \end{array}
\end{array}\right.
\end{array}\right) \\
\mathit{snd} \equiv \lambda\, h\, a\, c\, h' \;\rightarrow\; h\, a\, h'.\mathit{fst}\, h'.\mathit{snd}.\mathit{fst}\, c\, h'.\mathit{snd}.\mathit{snd}
\end{array}\right.
\end{array}
$$

4.14.15 Division LT-allegories of E-relations

Proposition

The distributive LT-allegory of setoids and their E-relations forms a division allegory, with left residual the left residual of E-relations.

Proof

$\mathit{SetoidDivAllegory} \in \mathit{DivisionAllegory}$

$$
\mathit{SetoidDivAllegory} \equiv \begin{array}{l}
\textbf{struct} \\
\left[\begin{array}{l}
\mathit{distalgy} \equiv \mathit{SetoidDistrAllegory} \\
\mathit{leftRes} \equiv /\!= \\
\mathit{defLeftRes} \equiv \lambda\, A\, B\, C\, R\, S\, X \;\rightarrow\; \mathit{isSolutIneqLeftRes}
\end{array}\right.
\end{array}
$$

4.14.16 Some kinds of allegories can't be constructed

Boolean LT-allegories of E-relations

As it turns out, setoids and E-relations do not form a Boolean allegory. This is because the law of double negation is not constructively valid. For the relational negation operator we are able to prove $R \subseteq \neg\neg R$ but not the other inequality, and so it would fail to be an involutive relation operator as would be necessary. This is because we can prove the propositional formula $A \rightarrow \neg\neg A$ but not the corresponding $\neg\neg A \rightarrow A$.

Power LT-allegories of E-relations

Setoids and E-relations do not form a power allegory. To formally prove this result is beyond the scope of this work, but we can justify it informally as follows. The power object is expected to be interpreted as the "powerset" of the original set, and should be a setoid itself. What would this powerset be? A subset of a set A would be a propositional function of type $A \to Set$. But this function space is not a set itself, it is a type (that is, it is an element of *Type*). In a predicative setting the powerset (a collection of all propositional functions of that type) would not be a set. Similarly, a subset of a setoid would be a propositional function that respects equality. And also similarly, a "powersetoid" would not be a setoid. A power object construction would thus produce an object of the next higher universe, instead of staying in the same universe as required.

For a more formal account of the impossibility of building a power allegory of E-sets and E-relations, we can first remark on the known fact that there is no analogue of the powerset axiom in constructive set theory. Also, as has been shown by Maietti and Valentini [51], extending Martin-Löf's type theory with a power set construction turns the extended theory into a classical one and we loose the possibility to exhibit proof elements.

4.15 LT-allegories of finite decidable relations

This kind of relations are important in practice, since many applications (e.g. database theory) are about finite decidable relations rather than general relations.

4.15.1 Introduction

We follow here a path quite similar to the one of the immediately preceding chapter, but now deal with LT-allegories of a restricted kind: we will have finite setoids and Boolean-valued equality preserving relations between them. In 9.1 we define the precategory of finite setoids and these decidable relations, and then (9.2-3) we show that we have a preorder and an equivalence comparisons between decidable relations. It is important to note that many of the manipulations follow closely those of the preceding chapter, after adjusting for "lifting" Boolean values to sets.

After this, 9.4-5 present the basic properties of composition and identity decidable relations, and put it all together in the LT-category of finite setoids and decidable relations. In the preceding section we carried out proofs by accessing the components of a proof object directly. In this section, however, this approach is not possible since proof objects for Boolean-valued propositions have no structure, as they come from raising Boolean values. The Alfa library provides elimination and introduction functions for all the Boolean-valued logical constants, and in particular for computable finite quantifiers and their properties. All these library elements will be present in much of the following material, since by definition composition of decidable relations uses the finite existential quantifier and intersection uses the Boolean and operation. 9.6 deals with the properties of converse and 9.7 with those of intersection of decidable relations, after which 9.8 puts it all together into the LT-allegory of finite setoids and decidable relations.

Then in 9.9 the properties of union of decidable relations are presented, making use of elimination and introduction constants for the Boolean or operation that defines union. 9.10 does the same for the empty decidable relation, and finally 9.11 puts it all together to form the distributive LT-allegory of finite setoids and decidable relations. Then in a new section (9.12) we show how complement of decidable relations satisfies the needed properties for forming a Boolean LT-allegory (9.13).

The last part of the chapter (9.14) shows that it is possible to have power allegories of finite setoids and decidable relations. The key for achieving this is by encoding subsets of a set by means of their characteristic function, in such a way that an element of a finite set of cardinality 2^n represents a subset of a finite set of size n by a direct binary encoding. We present all the lengthy details involved in such an encoding construction.

4.15.2 Precategories of decidable relations

Finite setoids and decidable relations over them form a precategory. Its objects are the finite setoids (here represented by their cardinality), its arrows Boolean E-relations between them, the preorder being inclusion between Boolean E-relations, identity arrows being the identities of the finite setoids, and arrow composition being composition of Boolean E-relations.

$FinSetoidPrecat \in LargePrecat$

$$FinSetoidPrecat \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} obj \equiv Nat \\ arr \equiv \lambda\, m\, n \;\rightarrow\; BoolERel\ (FinSetoid.Fin\, m)\ (FinSetoid.Fin\, n) \\ \subseteq \equiv \lambda\, m\, n \;\rightarrow \subseteq \\ I \equiv \lambda\, n \;\rightarrow\; I\, n \\ \bullet \equiv \lambda\, m\, n\, p \;\rightarrow\; \bullet = \end{array} \right. \end{array}$$

4.15.3 Inclusion of decidable relations is a preorder

Reflexivity

Proposition Inclusion of decidable relations is reflexive.

Proof

$$isReflLTBoolRel\ (R \in BoolHetRel\, A\, B) \in R \subseteq R \qquad \left(\begin{array}{l} A \in Set \\ B \in Set \end{array} \right)$$

$$isReflLTBoolRel\, R \equiv \lambda\, a\, b \;\rightarrow\; reflImp\ (|\, R\, a\, b\, |)$$

Transitivity

Proposition Inclusion of decidable relations is transitive.

Proof

$$isTransLTBoolRel \in (R \subseteq S) \to (S \subseteq T) \to R \subseteq T \qquad \left(\begin{array}{l} A \in Set \\ B \in Set \\ R \in BoolHetRel\, A\, B \\ S \in BoolHetRel\, A\, B \\ T \in BoolHetRel\, A\, B \end{array}\right)$$

$$\begin{array}{l} isTransLTBoolRel \equiv \lambda\, hRS\, hST\, a\, b \to \\ tranImp\ (|\, R\, a\, b\,|)\ (|\, S\, a\, b\,|)\ (|\, T\, a\, b\,|)\ (hRS\, a\, b)\ (hST\, a\, b) \end{array}$$

4.15.4 Equality of decidable relations is an equivalence relation

Reflexivity

Proposition Equality of decidable relations is reflexive.

Proof

$$isReflEqBoolRel\ (R \in BoolHetRel\, A\, B) \in R = R \qquad \left(\begin{array}{l} A \in Set \\ B \in Set \end{array}\right)$$

$$isReflEqBoolRel\ R \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv isReflLTBoolRel\ R \\ snd \equiv isReflLTBoolRel\ R \end{array}\right. \end{array}$$

Symmetry

Proposition Equality of decidable relations is symmetrical.

Proof

$$isSymmEqBoolRel \in (R = S) \to S = R \qquad \left(\begin{array}{l} A \in Set \\ B \in Set \\ R \in BoolHetRel\, A\, B \\ S \in BoolHetRel\, A\, B \end{array}\right)$$

$$isSymmEqBoolRel \equiv \lambda\, h \;\rightarrow\; \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv h.snd \\ snd \equiv h.fst \end{array}\right.\end{array}$$

Transitivity

Proposition Equality of decidable relations is transitive.

Proof

$$isTransEqBoolRel \;\in (R = S) \;\rightarrow\; (S = T) \;\rightarrow\; R = T \qquad \left(\begin{array}{l} A \in Set \\ B \in Set \\ R \in BoolHetRel\, A\, B \\ S \in BoolHetRel\, A\, B \\ T \in BoolHetRel\, A\, B \end{array}\right)$$

$$isTransEqBoolRel \equiv \lambda\, hRS\, hST \;\rightarrow\; \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv isTransLTBoolRel\, hRS.fst\, hST.fst \\ snd \equiv isTransLTBoolRel\, hST.snd\, hRS.snd \end{array}\right.\end{array}$$

4.15.5 Properties of composition

Associativity

Proposition Composition of decidable relations over finite sets is associative.

Proof

$$isAssocComposBoolRel \;\in ((R \bullet S) \bullet T) = (R \bullet (S \bullet T))$$
$$\left(\begin{array}{l} m \in Nat \\ n \in Nat \\ p \in Nat \\ q \in Nat \\ R \in BoolHetRel\ (Fin\, m)\ (Fin\, n) \\ S \in BoolHetRel\ (Fin\, n)\ (Fin\, p) \\ T \in BoolHetRel\ (Fin\, p)\ (Fin\, q) \end{array}\right)$$

$isAssocComposBoolRel \equiv$

```
struct
  ⎡ fst ≡ λ a d h →
  ⎢            ⎡ aux ∈ ∃c ∈ Fin p . | • R S a c ∧ T c d |
  ⎢            ⎣   aux ≡ (spec_exist p λ b → • R S a b ∧ T b d) .fst h
  ⎢            ⎡ aux′ ∈ ∃b ∈ Fin n . | R a b ∧ S b aux.fst |
  ⎢            ⎢   aux′ ≡ (spec_exist n λ b → R a b ∧ S b aux.fst) .fst
  ⎢       let  ⎢       ⎛ elim_and_fst                              ⎞
  ⎢            ⎢       ⎜   (∃b ∈ Fin n . R a b ∧ S b aux.fst)      ⎟
  ⎢            ⎢       ⎜   (T aux.fst d)                           ⎟
  ⎢            ⎣       ⎝   aux.snd                                 ⎠
  ⎢       in intro_exist n λ b → R a b ∧ • S T b d aux′.fst
  ⎢          (intro_and (R a aux′.fst) (• S T aux′.fst d)
  ⎢             (elim_and_fst (R a aux′.fst) (S aux′.fst aux.fst) aux′.snd)
  ⎢             (intro_exist p λ b → S aux′.fst b ∧ T b d aux.fst
  ⎢                (intro_and (S aux′.fst aux.fst) (T aux.fst d)
  ⎢                   (elim_and_snd (R a aux′.fst) (S aux′.fst aux.fst) aux′.snd)
  ⎣                   (elim_and_snd (• R S a aux.fst) (T aux.fst d) aux.snd))))
  ⎡ snd ≡ λ a d h →
  ⎢            ⎡ aux ∈ ∃b ∈ Fin n . | R a b ∧ • S T b d |
  ⎢            ⎢   aux ≡ (spec_exist n λ b → R a b ∧ • S T b d) .fst h
  ⎢            ⎢ aux′ ∈ ∃c ∈ Fin p . | S aux.fst c ∧ T c d |
  ⎢            ⎢   aux′ ≡ (spec_exist p λ c → S aux.fst c ∧ T c d) .fst
  ⎢       let  ⎢       ⎛ elim_and_snd                              ⎞
  ⎢            ⎢       ⎜   (R a aux.fst)                           ⎟
  ⎢            ⎢       ⎜   (∃c ∈ Fin p . S aux.fst c ∧ T c d)      ⎟
  ⎢            ⎣       ⎝   aux.snd                                 ⎠
  ⎢       in intro_exist p λ b arrow • R S a b ∧ T b d aux′.fst
  ⎢          (intro_and (• R S a aux′.fst) (T aux′.fst d)
  ⎢             (intro_exist n λ b arrow R a b ∧ S b aux′.fst aux.fst
  ⎢                (intro_and (R a aux.fst) (S aux.fst aux′.fst)
  ⎢                   (elim_and_fst (R a aux.fst) (• S T aux.fst d) aux.snd)
  ⎢                   (elim_and_fst (S aux.fst aux′.fst) (T aux′.fst d) aux′.snd)))
  ⎣             (elim_and_snd (S aux.fst aux′.fst) (T aux′.fst d) aux′.snd))
```

Monotonicity

Proposition Composition of decidable relations over finite sets is monotonic with respect to inclusion.

Proof

$$isMonotComposBoolRel \in (R1 \subseteq R2) \to (S1 \subseteq S2) \to (R1 \bullet S1) \subseteq (R2 \bullet S2)$$
$$\left(\begin{array}{l} m \in Nat \\ n \in Nat \\ p \in Nat \\ R1 \in BoolHetRel\ (Fin\, m)\ (Fin\, n) \\ R2 \in BoolHetRel\ (Fin\, m)\ (Fin\, n) \\ S1 \in BoolHetRel\ (Fin\, n)\ (Fin\, p) \\ S2 \in BoolHetRel\ (Fin\, n)\ (Fin\, p) \end{array}\right)$$

$$\begin{array}{l} isMonotComposBoolRel \equiv \lambda\, hR\, hS\, a\, c\, h \to \\ \quad \mathbf{let} \left[\begin{array}{l} aux \in \exists\, b \in Fin\, n\, .\, |R1\, a\, b \wedge S1\, b\, c| \\ \quad aux \equiv (spec\_exist\, n\, \lambda\, b \to R1\, a\, b \wedge S1\, b\, c)\, .fst\, h \end{array} \right. \\ \quad \mathbf{in}\, intro\_exist\, n\, \lambda\, b \to R2\, a\, b \wedge S2\, b\, c\, aux.fst \\ \qquad (intro\_and\ (R2\, a\, aux.fst)\ (S2\, aux.fst\, c) \\ \qquad\quad (hR\, a\, aux.fst\ (elim\_and\_fst\ (R1\, a\, aux.fst)\ (S1\, aux.fst\, c)\ aux.snd)) \\ \qquad\quad (hS\, aux.fst\, c\ (elim\_and\_snd\ (R1\, a\, aux.fst)\ (S1\, aux.fst\, c)\ aux.snd))) \end{array}$$

Right identity

Proposition The identity relation on a finite set is a right identity for decidable relation composition.

Proof

$$isRightIdBool \in (R \bullet = I\, n) = R$$
$$\left(\begin{array}{l} m \in Nat \\ n \in Nat \\ R \in BoolERel\ (FinSetoid.Fin\, m)\ (FinSetoid.Fin\, n) \end{array}\right)$$

$$\begin{array}{l} isRightIdBool \equiv \\ \quad \mathbf{struct} \\ \qquad \left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h \to \\ \qquad\qquad \mathbf{let} \left[\begin{array}{l} aux \in \exists\, b' \in Fin\, n\, .\, |R.rel\, a\, b' \wedge (I\, n)\, .rel\, b'\, b| \\ \quad aux \equiv (spec\_exist\, n\, \lambda\, b' \to R.rel\, a\, b' \wedge (I\, n)\, .rel\, b'\, b)\, .fst\, h \end{array} \right. \\ \qquad\qquad \mathbf{in}\, R.resp\, a\, a\, aux.fst\, b\ (ref\ (FinSetoid.Fin\, m)\ a) \\ \qquad\qquad\quad (elim\_and\_snd\ (R.rel\, a\, aux.fst)\ ((I\, n)\, .rel\, aux.fst\, b)\ aux.snd) \\ \qquad\qquad\quad (elim\_and\_fst\ (R.rel\, a\, aux.fst)\ ((I\, n)\, .rel\, aux.fst\, b)\ aux.snd) \end{array} \right. \\ \qquad \left[\begin{array}{l} snd \equiv \lambda\, a\, b\, h \to intro\_exist\, n\, \lambda\, b' \to R.rel\, a\, b' \wedge (I\, n)\, .rel\, b'\, b\, b \\ \qquad\qquad (intro\_and\ (R.rel\, a\, b)\ ((I\, n)\, .rel\, b\, b)\ h\ (ref\ (FinSetoid.Fin\, n)\ b)) \end{array} \right. \end{array}$$

Left identity

Proposition The identity relation on a finite set is a left identity for decidable relation composition.

Proof

$$isLeftIdBool \in (I\,m \bullet = R) = R$$
$$\left(\begin{array}{l} m \in Nat \\ n \in Nat \\ R \in BoolERel\ (FinSetoid.Fin\,m)\ (FinSetoid.Fin\,n) \end{array}\right)$$

$$\begin{array}{l}
isLeftIdBool \equiv \\
\quad \textbf{struct} \\
\quad\quad \left[\begin{array}{l}
fst \equiv \lambda\, a\, b\, h \rightarrow \\
\quad\quad \textbf{let} \left[\begin{array}{l} aux \in \exists a' \in Fin\,m\,.\ |\,(I\,m).rel\,a\,a' \wedge R.rel\,a'\,b\,| \\ aux \equiv (spec_exist\,m\,\lambda\,b' \rightarrow (I\,m).rel\,a\,b' \wedge R.rel\,b'\,b).fst\,h \end{array}\right. \\
\quad\quad \textbf{in}\ R.resp\,aux.fst\,a\,b\,b \\
\quad\quad\quad (sym\,(FinSetoid.Fin\,m)\,a\,aux.fst \\
\quad\quad\quad\quad (elim_and_fst\,((I\,m).rel\,a\,aux.fst)\,(R.rel\,aux.fst\,b)\,aux.snd)) \\
\quad\quad\quad (ref\,(FinSetoid.Fin\,n)\,b) \\
\quad\quad\quad (elim_and_snd\,((I\,m).rel\,a\,aux.fst)\,(R.rel\,aux.fst\,b)\,aux.snd)
\end{array}\right. \\
\quad\quad \left[\begin{array}{l}
snd \equiv \lambda\, a\, b\, h \rightarrow intro_exist\,m\,\lambda\,b' \rightarrow (I\,m).rel\,a\,b' \wedge R.rel\,b'\,b\,a \\
\quad\quad (intro_and\,((I\,m).rel\,a\,a)\,(R.rel\,a\,b)\,(ref\,(FinSetoid.Fin\,m)\,a)\,h)
\end{array}\right.
\end{array}$$

4.15.6 LT-categories of decidable relations

Proposition

The precategory of finite setoids and their Boolean E-relations forms an LT-category.

Proof

$$FinSetoidLTCat \in LargeLTCat$$

$$FinSetoidLTCat \equiv \begin{array}{l}
\textbf{struct} \\
\left[\begin{array}{l}
pcat \equiv FinSetoidPrecat \\
ltIsRef \equiv \lambda\, m\, n\, R \rightarrow isReflLTBoolRel\ R.rel \\
ltIsTra \equiv \lambda\, m\, n\, R\, S\, T \rightarrow isTransLTBoolRel \\
compIsMonot \equiv \lambda\, m\, n\, p\, R1\, R2\, S1\, S2 \rightarrow isMonotComposBoolRel \\
isLId \equiv \lambda\, m\, n\, R \rightarrow isLeftIdBool \\
isRId \equiv \lambda\, m\, n\, R \rightarrow isRightIdBool \\
isAso \equiv \lambda\, m\, n\, p\, q\, R\, S\, T \rightarrow isAssocComposBoolRel
\end{array}\right.
\end{array}$$

4.15.7 Properties of converse

Involution

Proposition Converse of decidable relations is involutive.

Proof

$$isInvolConvBoolRel \in ((R^\circ)^\circ) = R \qquad \left(\begin{array}{l} A \in Set \\ B \in Set \\ R \in BoolHetRel\, A\, B \end{array}\right)$$

$$isInvolConvBoolRel \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h \rightarrow h \\ snd \equiv \lambda\, a\, b\, h \rightarrow h \end{array}\right. \end{array}$$

Order preservation

Proposition Converse of decidable relations is order preserving with respect to relation inclusion.

Proof

$$isOrdPresConvBoolRel \in (R \subseteq S) \leftrightarrow ((R^\circ) \subseteq (S^\circ)) \qquad \left(\begin{array}{l} A \in Set \\ B \in Set \\ R \in BoolHetRel\, A\, B \\ S \in BoolHetRel\, A\, B \end{array}\right)$$

$$isOrdPresConvBoolRel \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv \lambda\, h\, a\, b\, h' \rightarrow h\, b\, a\, h' \\ snd \equiv \lambda\, h\, a\, b\, h' \rightarrow h\, b\, a\, h' \end{array}\right. \end{array}$$

Contravariance

Proposition Converse of decidable relations over finite sets is contravariant with respect to composition of such relations.

Proof

$$isContravConvBoolRel \in ((R \bullet S)^\circ) = ((S^\circ) \bullet (R^\circ))$$
$$\left(\begin{array}{l} m \in Nat \\ n \in Nat \\ p \in Nat \\ R \in BoolHetRel\ (Fin\, m)\ (Fin\, n) \\ S \in BoolHetRel\ (Fin\, n)\ (Fin\, p) \end{array}\right)$$

$$\begin{array}{l} isContravConvBoolRel \equiv \\ \ \ \textbf{struct} \\ \quad \left[\ fst \equiv \lambda\, c\, a\, h \ \rightarrow \begin{array}{l} \textbf{let} \left[\begin{array}{l} aux \in \exists\, b \in Fin\, n\, .\, |\, R\, a\, b \wedge S\, b\, c\, | \\ \quad aux \equiv (spec\_exist\, n\, \lambda\, b \rightarrow R\, a\, b \wedge S\, b\, c)\, .fst\, h \end{array}\right. \\ \textbf{in}\ intro\_exist\, n\, \lambda\, b \rightarrow {}^\circ S\, c\, b \wedge {}^\circ R\, b\, a\, aux.fst \\ \quad (intro\_and\, (S\, aux.fst\, c)\, (R\, a\, aux.fst) \\ \qquad (elim\_and\_snd\, (R\, a\, aux.fst)\, (S\, aux.fst\, c)\, aux.snd) \\ \qquad (elim\_and\_fst\, (R\, a\, aux.fst)\, (S\, aux.fst\, c)\, aux.snd)) \end{array}\right. \\ \quad \left[\ snd \equiv \lambda\, c\, a\, h \ \rightarrow \begin{array}{l} \textbf{let} \left[\begin{array}{l} aux \in \exists\, b \in Fin\, n\, .\, |\, {}^\circ S\, c\, b \wedge {}^\circ R\, b\, a\, | \\ \quad aux \equiv (spec\_exist\, n\, \lambda\, b \rightarrow {}^\circ S\, c\, b \wedge {}^\circ R\, b\, a)\, .fst\, h \end{array}\right. \\ \textbf{in}\ intro\_exist\, n\, \lambda\, b \rightarrow R\, a\, b \wedge S\, b\, c\, aux.fst \\ \quad (intro\_and\ (R\, a\, aux.fst)\ (S\, aux.fst\, c) \\ \qquad (elim\_and\_snd\ (S\, aux.fst\, c)\ (R\, a\, aux.fst)\ aux.snd) \\ \qquad (elim\_and\_fst\ (S\, aux.fst\, c)\ (R\, a\, aux.fst)\ aux.snd)) \end{array}\right. \end{array}$$

4.15.8 Properties of intersection

Greatest lower bound

Proposition Intersection of decidable relations is the greatest lower bound of its argument relations with respect to relation inclusion.

Proof

$$isGLBInterBoolRel \in (X \subseteq (R \cap S)) \leftrightarrow ((X \subseteq R)\ \&\ (X \subseteq S))$$
$$\left(\begin{array}{l} A \in Set \\ B \in Set \\ R \in BoolHetRel\, A\, B \\ S \in BoolHetRel\, A\, B \\ X \in BoolHetRel\, A\, B \end{array}\right)$$

$$\begin{array}{l} isGLBInterBoolRel \equiv \\ \ \ \textbf{struct} \\ \quad \left[\ fst \equiv \lambda\, h \rightarrow \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h' \rightarrow elim\_and\_fst\ (R\, a\, b)\ (S\, a\, b)\ (h\, a\, b\, h') \\ snd \equiv \lambda\, a\, b\, h' \rightarrow elim\_and\_snd\ (R\, a\, b)\ (S\, a\, b)\ (h\, a\, b\, h') \end{array}\right. \end{array}\right. \\ \quad \left[\ snd \equiv \lambda\, h\, a\, b\, h' \rightarrow intro\_and\ (R\, a\, b)\ (S\, a\, b)\ (h.fst\, a\, b\, h')\ (h.snd\, a\, b\, h')\right. \end{array}$$

Modular law

Proposition For decidable relations over finite sets, the modular law holds.

Proof

$$\mathit{modularLawBool} \in ((R \bullet S) \cap T) \subseteq ((R \cap (T \bullet (S^{\circ}))) \bullet S)$$
$$\begin{pmatrix} m \in \mathit{Nat} \\ n \in \mathit{Nat} \\ p \in \mathit{Nat} \\ R \in \mathit{BoolHetRel}\ (\mathit{Fin}\, m)\ (\mathit{Fin}\, n) \\ S \in \mathit{BoolHetRel}\ (\mathit{Fin}\, n)\ (\mathit{Fin}\, p) \\ T \in \mathit{BoolHetRel}\ (\mathit{Fin}\, m)\ (\mathit{Fin}\, p) \end{pmatrix}$$

$$\begin{array}{l}
\mathit{modularLawBool} \equiv \\
\lambda\, a\, c\, h \to \\
\quad \textbf{let} \left[\begin{array}{l} \mathit{aux} \in \exists b \in \mathit{Fin}\, n\, .\, |R\, a\, b \wedge S\, b\, c| \\ \quad \mathit{aux} \equiv (\mathit{spec_exist}\, n\, \lambda\, b \to R\, a\, b \wedge S\, b\, c).\mathit{fst} \\ \qquad\qquad\qquad (\mathit{elim_and_fst}\, (\bullet\, R\, S\, a\, c)\, (T\, a\, c)\, h) \end{array} \right. \\
\quad \textbf{in}\, \mathit{intro_exist}\, n\, \lambda\, b \to \cap R\, (T \bullet (S^{\circ}))\, a\, b \wedge S\, b\, c\, \mathit{aux.fst} \\
\qquad (\mathit{intro_and}\, (\cap R\, (T \bullet (S^{\circ}))\, a\, \mathit{aux.fst})\, (S\, \mathit{aux.fst}\, c) \\
\qquad\quad (\mathit{intro_and}\, (R\, a\, \mathit{aux.fst})\, (\bullet\, T\, (S^{\circ})\, a\, \mathit{aux.fst}) \\
\qquad\qquad (\mathit{elim_and_fst}\, (R\, a\, \mathit{aux.fst})\, (S\, \mathit{aux.fst}\, c)\, \mathit{aux.snd}) \\
\qquad\qquad (\mathit{intro_exist}\, p\, \lambda\, b \to T\, a\, b \wedge {}^{\circ}\, S\, b\, \mathit{aux.fst}\, c \\
\qquad\qquad\quad (\mathit{intro_and}\, (T\, a\, c)\, ({}^{\circ}\, S\, c\, \mathit{aux.fst})\, (\mathit{elim_and_snd}\, (\bullet\, R\, S\, a\, c)\, (T\, a\, c)\, h) \\
\qquad\qquad\qquad (\mathit{elim_and_snd}\, (R\, a\, \mathit{aux.fst})\, (S\, \mathit{aux.fst}\, c)\, \mathit{aux.snd})))) \\
\qquad\quad (\mathit{elim_and_snd}\, (R\, a\, \mathit{aux.fst})\, (S\, \mathit{aux.fst}\, c)\, \mathit{aux.snd}))
\end{array}$$

4.15.9 LT-allegories of decidable relations

Proposition

The LT-category of finite setoids and their Boolean E-relations form an LT-allegory, with meet equal to intersection of relations, and converse equal to converse of relations.

Proof

$$\mathit{FinSetoidAllegory} \in \mathit{LTAllegory}$$

$$FinSetoidAllegory \equiv \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} ltcat \equiv FinSetoidLTCat \\ \cap \equiv \lambda\, m\, n \ \rightarrow\ \cap = \\ \circ \equiv \lambda\, m\, n \ \rightarrow\ \circ = \\ defMeet \equiv \lambda\, m\, n\, R\, S\, X \ \rightarrow\ isGLBInterBoolRel \\ isInvolConv \equiv \lambda\, m\, n\, R \ \rightarrow\ isInvolConvBoolRel \\ isOrdPresConv \equiv \lambda\, m\, n\, R\, S \ \rightarrow\ isOrdPresConvBoolRel \\ isContravConv \equiv \lambda\, m\, n\, p\, R\, S \ \rightarrow\ isContravConvBoolRel \\ modLaw \equiv \lambda\, m\, n\, p\, R\, S\, T \ \rightarrow\ modularLawBool \end{array}\right.\end{array}$$

4.15.10 Properties of union

Least upper bound

$$isLUBUnionBoolRel \in ((R \cup S) \subseteq X) \leftrightarrow ((R \subseteq X) \,\&\, (S \subseteq X))$$
$$\left(\begin{array}{l} A \in Set \\ B \in Set \\ R \in BoolHetRel\, A\, B \\ S \in BoolHetRel\, A\, B \\ X \in BoolHetRel\, A\, B \end{array}\right)$$

$$isLUBUnionBoolRel \equiv$$
$$\begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv \lambda\, h \ \rightarrow\ \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h' \ \rightarrow\ h\, a\, b\ (intro\_or\_lft\ (R\, a\, b)\ (S\, a\, b)\ h') \\ snd \equiv \lambda\, a\, b\, h' \ \rightarrow\ h\, a\, b\ (intro\_or\_rgt\ (R\, a\, b)\ (S\, a\, b)\ h') \end{array}\right.\end{array} \\ snd \equiv \lambda\, h\, a\, b\, h' \ \rightarrow\ when\_or\ (R\, a\, b)\ (S\, a\, b) \\ \qquad (\lambda\, hR \ \rightarrow\ h.fst\, a\, b\, hR)\, (\lambda\, hS \ \rightarrow\ h.snd\, a\, b\, hS)\, h' \end{array}\right.\end{array}$$

Distributivity of intersection

Proposition Intersection of decidable relations distributes over union of decidable relations.

Proof

$$isDistrBoolIntersUnion \in (R \cap (S \cup T)) = ((R \cap S) \cup (R \cap T))$$
$$\begin{pmatrix} A \in Set \\ B \in Set \\ R \in BoolHetRel\ A\ B \\ S \in BoolHetRel\ A\ B \\ T \in BoolHetRel\ A\ B \end{pmatrix}$$

$$\begin{array}{l}
isDistrBoolIntersUnion \equiv \\
\ \mathbf{struct} \\
\quad \left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h \;\rightarrow\; when_or\ (S\,a\,b)\ (T\,a\,b) \\
\qquad\quad \lambda\, hS \;\rightarrow\; intro_or_lft\ (\cap\, R\, S\, a\, b)\ (\cap\, R\, T\, a\, b) \\
\qquad\qquad (intro_and\ (R\,a\,b)\ (S\,a\,b)\ (elim_and_fst\ (R\,a\,b)\ (\cup\, S\, T\, a\, b)\ h)\ hS) \\
\qquad\quad \lambda\, hT \;\rightarrow\; intro_or_rgt\ (\cap\, R\, S\, a\, b)\ (\cap\, R\, T\, a\, b) \\
\qquad\qquad (intro_and\ (R\,a\,b)\ (T\,a\,b)\ (elim_and_fst\ (R\,a\,b)\ (\cup\, S\, T\, a\, b)\ h)\ hT) \\
\qquad\qquad\quad (elim_and_snd\ (R\,a\,b)\ (\cup\, S\, T\, a\, b)\ h) \end{array} \right. \\
\quad \left[\begin{array}{l} snd \equiv \lambda\, a\, b\, h \;\rightarrow\; when_or\ (\cap\, R\, S\, a\, b)\ (\cap\, R\, T\, a\, b) \\
\qquad\quad \lambda\, hRS \;\rightarrow\; intro_and\ (R\,a\,b)\ (\cup\, S\, T\, a\, b)\ (elim_and_fst\ (R\,a\,b)\ (S\,a\,b)\ hRS) \\
\qquad\qquad (intro_or_lft\ (S\,a\,b)\ (T\,a\,b)\ (elim_and_snd\ (R\,a\,b)\ (S\,a\,b)\ hRS)) \\
\qquad\quad \lambda\, hRT \;\rightarrow\; intro_and\ (R\,a\,b)\ (\cup\, S\, T\, a\, b)\ (elim_and_fst\ (R\,a\,b)\ (T\,a\,b)\ hRT) \\
\qquad\qquad (intro_or_rgt\ (S\,a\,b)\ (T\,a\,b)\ (elim_and_snd\ (R\,a\,b)\ (T\,a\,b)\ hRT))\ h \end{array} \right.
\end{array}$$

Distributivity of composition

Proposition Composition of decidable relations over finite sets distributes over union of such relations.

Proof

$$isDistrBoolComposUnion \in (R \bullet (S \cup T)) = ((R \bullet S) \cup (R \bullet T))$$
$$\begin{pmatrix} m \in Nat \\ n \in Nat \\ p \in Nat \\ R \in BoolHetRel\ (Fin\ m)\ (Fin\ n) \\ S \in BoolHetRel\ (Fin\ n)\ (Fin\ p) \\ T \in BoolHetRel\ (Fin\ n)\ (Fin\ p) \end{pmatrix}$$

$$isDistrBoolComposUnion \equiv$$

struct

$$
\left[\begin{array}{l}
\mathit{fst} \equiv \lambda\, a\, c\, h \to \\
\quad \textbf{let} \left[\begin{array}{l}
\mathit{aux} \in \exists\, b \in \mathit{Fin}\, n\, .\, |R\, a\, b \wedge \cup\, S\, T\, b\, c| \\
\quad \mathit{aux} \equiv (\mathit{spec\_exist}\, n\, \lambda\, b \to R\, a\, b \wedge \cup\, S\, T\, b\, c).\mathit{fst}\, h
\end{array}\right. \\
\quad \textbf{in}\ \mathit{when\_or}\ (S\, \mathit{aux.fst}\, c)\ (T\, \mathit{aux.fst}\, c) \\
\qquad \lambda\, hS \to \mathit{intro\_or\_lft}\ (\bullet\, R\, S\, a\, c)\ (\bullet\, R\, T\, a\, c) \\
\qquad\quad (\mathit{intro\_exist}\, n\, \lambda\, b \to R\, a\, b \wedge S\, b\, c\, \mathit{aux.fst} \\
\qquad\qquad (\mathit{intro\_and}\ (R\, a\, \mathit{aux.fst})\ (S\, \mathit{aux.fst}\, c) \\
\qquad\qquad\quad (\mathit{elim\_and\_fst}\ (R\, a\, \mathit{aux.fst})\ (\cup\, S\, T\, \mathit{aux.fst}\, c)\ \mathit{aux.snd})\ hS)) \\
\qquad \lambda\, hT \to \mathit{intro\_or\_rgt}\ (\bullet\, R\, S\, a\, c)\ (\bullet\, R\, T\, a\, c) \\
\qquad\quad (\mathit{intro\_exist}\, n\, \lambda\, b \to R\, a\, b \wedge T\, b\, c\, \mathit{aux.fst} \\
\qquad\qquad (\mathit{intro\_and}\ (R\, a\, \mathit{aux.fst})\ (T\, \mathit{aux.fst}\, c) \\
\qquad\qquad\quad (\mathit{elim\_and\_fst}\ (R\, a\, \mathit{aux.fst})\ (\cup\, S\, T\, \mathit{aux.fst}\, c)\ \mathit{aux.snd})\ hT)) \\
\qquad\qquad\quad (\mathit{elim\_and\_snd}\ (R\, a\, \mathit{aux.fst})\ (\cup\, S\, T\, \mathit{aux.fst}\, c)\ \mathit{aux.snd})
\end{array}\right.
$$

$$
\left[\begin{array}{ll}
\mathit{snd} \equiv \lambda\, a\, c\, h \to \mathit{when\_or}\ (\bullet\, R\, S\, a\, c)\ (\bullet\, R\, T\, a\, c) & \\
 & \textbf{let} \left[\begin{array}{l}
\mathit{aux} \in \exists\, n' \in \mathit{Fin}\, n\, .\, |R\, a\, n' \wedge S\, n'\, c| \\
\quad \mathit{aux} \equiv (\mathit{spec\_exist}\, n\, \lambda\, b \to R\, a\, b \wedge S\, b\, c).\mathit{fst}\, hRS
\end{array}\right. \\
 & \textbf{in}\ \mathit{intro\_exist}\, n\, \lambda\, b \to R\, a\, b \wedge \cup\, S\, T\, b\, c\, \mathit{aux.fst} \\
\quad \lambda\, hRS \to & \quad (\mathit{intro\_and}\ (R\, a\, \mathit{aux.fst})\ (\cup\, S\, T\, \mathit{aux.fst}\, c) \\
 & \qquad (\mathit{elim\_and\_fst}\ (R\, a\, \mathit{aux.fst})\ (S\, \mathit{aux.fst}\, c)\ \mathit{aux.snd}) \\
 & \qquad (\mathit{intro\_or\_lft}\ (S\, \mathit{aux.fst}\, c)\ (T\, \mathit{aux.fst}\, c) \\
 & \qquad\quad (\mathit{elim\_and\_snd}\ (R\, a\, \mathit{aux.fst})\ (S\, \mathit{aux.fst}\, c)\ \mathit{aux.snd}))) \\
 & \textbf{let} \left[\begin{array}{l}
\mathit{aux} \in \exists\, n' \in \mathit{Fin}\, n\, .\, |R\, a\, n' \wedge T\, n'\, c| \\
\quad \mathit{aux} \equiv (\mathit{spec\_exist}\, n\, \lambda\, b \to R\, a\, b \wedge T\, b\, c).\mathit{fst}\, hRT
\end{array}\right. \\
 & \textbf{in}\ \mathit{intro\_exist}\, n\, \lambda\, b \to R\, a\, b \wedge \cup\, S\, T\, b\, c\, \mathit{aux.fst} \\
\quad \lambda\, hRT \to & \quad (\mathit{intro\_and}\ (R\, a\, \mathit{aux.fst})\ (\cup\, S\, T\, \mathit{aux.fst}\, c) \\
 & \qquad (\mathit{elim\_and\_fst}\ (R\, a\, \mathit{aux.fst})\ (T\, \mathit{aux.fst}\, c)\ \mathit{aux.snd}) \\
 & \qquad (\mathit{intro\_or\_rgt}\ (S\, \mathit{aux.fst}\, c)\ (T\, \mathit{aux.fst}\, c) \\
 & \qquad\quad (\mathit{elim\_and\_snd}\ (R\, a\, \mathit{aux.fst})\ (T\, \mathit{aux.fst}\, c)\ \mathit{aux.snd}))) \\
\quad h &
\end{array}\right.
$$

4.15.11 Properties of the empty relation

Zero for intersection

Proposition The empty relation is a zero on the right for intersection of decidable relations.

Proof

$$
\mathit{isZeroBoolInters} \in (R \cap \emptyset) = \emptyset \qquad \left(\begin{array}{l} A \in \mathit{Set} \\ B \in \mathit{Set} \\ R \in \mathit{BoolHetRel}\, A\, B \end{array}\right)
$$

$$isZeroBoolInters \equiv \quad \mathbf{struct} \left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h \rightarrow elim\_and\_snd\ (R\, a\, b)\ (\emptyset\, a\, b)\ h \\ snd \equiv \lambda\, a\, b\, h \rightarrow elimAbsurd\ (|\cap\ R\, \emptyset\, a\, b\, |)\ h \end{array}\right.$$

Zero for union

Proposition The empty relation is a zero on the right for union of decidable relations.

Proof

$$isZeroBoolUnion \in (R \cup \emptyset) = R \qquad \left(\begin{array}{l} A \in Set \\ B \in Set \\ R \in BoolHetRel\, A\, B \end{array}\right)$$

$$isZeroBoolUnion \equiv \quad \mathbf{struct} \left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h \rightarrow when\_or\ (R\, a\, b)\ (\emptyset\, a\, b) \\ \qquad\qquad \lambda\, hR \rightarrow hR \\ \qquad\qquad \lambda\, h' \rightarrow elimAbsurd\ (|\, R\, a\, b\, |)\ h' \\ \qquad\qquad h \\ snd \equiv \lambda\, a\, b\, h \rightarrow intro\_or\_lft\ (R\, a\, b)\ (\emptyset\, a\, b)\ h \end{array}\right.$$

Zero for composition

Proposition The empty relation is a zero on the right for composition of decidable relations on finite sets.

Proof

$$isZeroBoolCompos \in (R \bullet \emptyset) = \emptyset \qquad \left(\begin{array}{l} m \in Nat \\ n \in Nat \\ p \in Nat \\ R \in BoolHetRel\ (Fin\, m)\ (Fin\, n) \end{array}\right)$$

$$\begin{array}{l} isZeroBoolCompos \equiv \\ \mathbf{struct} \\ \left[\begin{array}{l} fst \equiv \lambda\, a\, c\, h \rightarrow \\ \qquad \mathbf{let} \left[\begin{array}{l} aux \in \exists\, b \in Fin\, n\, .\, |\, R\, a\, b \wedge \emptyset\, b\, c\, | \\ aux \equiv (spec\_exist\, n\, \lambda\, b \rightarrow R\, a\, b \wedge \emptyset\, b\, c).fst\, h \end{array}\right. \\ \qquad \mathbf{in}\ elimAbsurd\ (|\, \emptyset\, a\, c\, |)\ (elim\_and\_snd\ (R\, a\, aux.fst)\ (\emptyset\, aux.fst\, c)\ aux.snd) \\ snd \equiv \lambda\, a\, c\, h \rightarrow elimAbsurd\ (|\bullet\ R\, \emptyset\, a\, c\, |)\ h \end{array}\right. \end{array}$$

4.15.12 Distributive LT-allegories of decidable relations

Proposition

The LT-allegory of finite setoids and their decidable relations forms a distributive allegory, with union as its join, and the empty relation as the zero arrow.

Proof

$$\mathit{FinSetoidDistAllegory} \in \mathit{DistribAllegory}$$

$$\begin{array}{l}
\mathit{FinSetoidDistAllegory} \equiv \\
\quad \textbf{struct} \\
\quad\quad \left[\begin{array}{l}
\mathit{algy} \equiv \mathit{FinSetoidAllegory} \\
\cup \equiv \lambda\, m\, n \rightarrow \cup = \\
0 \equiv \lambda\, m\, n \rightarrow \emptyset = \\
\mathit{defJoin} \equiv \lambda\, m\, n\, R\, S\, X \rightarrow \mathit{isLUBUnionBoolRel} \\
\mathit{isDistrMeetJoin} \equiv \lambda\, m\, n\, R\, S\, T \rightarrow \mathit{isDistrBoolIntersUnion} \\
\mathit{isDistrComposJoin} \equiv \lambda\, m\, n\, p\, R\, S\, T \rightarrow \mathit{isDistrBoolComposUnion} \\
\mathit{isZeroMeet} \equiv \lambda\, m\, n\, R \rightarrow \mathit{isZeroBoolInters} \\
\mathit{isZeroJoin} \equiv \lambda\, m\, n\, R \rightarrow \mathit{isZeroBoolUnion} \\
\mathit{isZeroCompos} \equiv \lambda\, m\, n\, p\, R \rightarrow \mathit{isZeroBoolCompos}
\end{array}\right.
\end{array}$$

4.15.13 Properties of complement

Order reversing

Proposition Complement of decidable relations is order reversing with respect to inclusion of relations.

Proof

$$\mathit{isOrdRevBoolNeg} \in (R \subseteq S) \rightarrow \neg\, S \subseteq \neg\, R \qquad \left(\begin{array}{l} A \in \mathit{Set} \\ B \in \mathit{Set} \\ R \in \mathit{BoolHetRel}\, A\, B \\ S \in \mathit{BoolHetRel}\, A\, B \end{array}\right)$$

$$\begin{array}{l}
\mathit{isOrdRevBoolNeg} \equiv \lambda\, h\, a\, b\, h' \rightarrow \mathit{intro_not}\ (R\, a\, b)\ \lambda\, hR \rightarrow \\
\quad\quad \mathit{elim_not}\ (S\, a\, b)\ h'\ (h\, a\, b\, hR)
\end{array}$$

De Morgan law

Proposition Complement of decidable relations satisfies the De Morgan law.

Proof

$$deMorganBoolNeg \in \neg\,(R \cup S) = (\neg\, R \cap \neg\, S) \qquad \left(\begin{array}{l} A \in Set \\ B \in Set \\ R \in BoolHetRel\; A\; B \\ S \in BoolHetRel\; A\; B \end{array}\right)$$

$deMorganBoolNeg \equiv$
struct

$$\left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h \;\rightarrow\; intro\_and\, (\neg\, (R\, a\, b))\, (\neg\, (S\, a\, b)) \\ \quad (intro\_not\, (R\, a\, b) \\ \quad\quad \lambda\, h' \;\rightarrow\; (spec\_not\, (R\, a\, b \;\vee\; S\, a\, b)).fst\, h\, (intro\_or\_lft\, (R\, a\, b)\, (S\, a\, b)\, h')) \\ \quad (intro\_not\, (S\, a\, b) \\ \quad\quad \lambda\, h' \;\rightarrow\; (spec\_not\, (R\, a\, b \;\vee\; S\, a\, b)).fst\, h\, (intro\_or\_rgt\, (R\, a\, b)\, (S\, a\, b)\, h')) \end{array}\right.$$

$$\left[\begin{array}{l} snd \equiv \lambda\, a\, b\, h \;\rightarrow\; when\_and\; (\neg\; (R\, a\, b))\; (\neg\; (S\, a\, b)) \\ \qquad\quad \lambda\, hNR\, hNS \;\rightarrow\; intro\_not\; (R\, a\, b \;\vee\; S\, a\, b) \\ \qquad\qquad \lambda\, h' \;\rightarrow\; elimOr\; (|\, R\, a\, b\, |)\; (|\, S\, a\, b\, |)\; \bot \\ \qquad\qquad\quad \lambda\, hR \;\rightarrow\; elim\_not\; (R\, a\, b)\; hNR\, hR \\ \qquad\qquad\quad \lambda\, hS \;\rightarrow\; elim\_not\; (S\, a\, b)\; hNS\, hS \\ \qquad\qquad\qquad ((spec\_or\; (R\, a\, b)\; (S\, a\, b))\, .fst\, h')\; h \end{array}\right.$$

Involution

Proposition Complement of decidable relations is an involution.

Proof

$$isInvolBoolNeg \in \neg\,(\neg\, R) = R \qquad \left(\begin{array}{l} A \in Set \\ B \in Set \\ R \in BoolHetRel\; A\; B \end{array}\right)$$

$isInvolBoolNeg \equiv$
struct

$$\left[\begin{array}{l} fst \equiv \lambda\, a\, b\, h \;\rightarrow \\ \qquad\quad \mathbf{case}\; decTrue\; (R\, a\, b)\;\; \mathbf{of} \\ \qquad\qquad \mathbf{inl}\, inR \;\rightarrow\; inR \\ \qquad\qquad \mathbf{inr}\, notR \;\rightarrow\; elimAbsurd\; (|\, R\, a\, b\, |) \\ \qquad\qquad\qquad (elim\_not\; (\neg\; (R\, a\, b))\; h\; ((spec\_not\; (R\, a\, b))\, .snd\, notR)) \end{array}\right.$$

$$\Big[\; snd \equiv \lambda\, a\, b\, h \;\rightarrow\; intro\_not\; (\neg\; (R\, a\, b))\; \lambda\, h' \;\rightarrow\; elim\_not\; (R\, a\, b)\; h'\, h$$

4.15.14 Boolean LT-allegories of decidable relations

Proposition

The distributive LT-allegory of finite sets and decidable relations over them forms a Boolean LT-allegory. Complement of relations is the negation of this allegory.

Proof

$FinSetoidBoolAllegory \in BooleanAllegory$

$$FinSetoidBoolAllegory \equiv \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} distalgy \equiv FinSetoidDistAllegory \\ neg \equiv \lambda\, m\, n \;\rightarrow\; \neg = \\ isOrdRevNeg \equiv \lambda\, A\, B\, R\, S \;\rightarrow\; isOrdRevBoolNeg \\ deMorgan \equiv \lambda\, A\, B\, R\, S \;\rightarrow\; deMorganBoolNeg \\ isInvolNeg \equiv \lambda\, A\, B\, R \;\rightarrow\; isInvolBoolNeg \end{array}\right.\end{array}$$

4.15.15 Power LT-allegories of finite decidable relations

The key idea for representing the different components of a power allegory is to use *characteristic functions* for the subsets of a powerset. Such a function takes an element of the original set and gives a result of 1 or 0 according to the presence or absence of the element in the subset represented by that characteristic function. Clearly, we can think of the elements of $Fin\ n$ as being the natural numbers $0, \ldots, n-1$. The power object of the setoid based on $Fin\ n$ is simply the setoid based on $Fin\ 2^n$. We view an element x of $Fin\ 2^n$ as an encoding in binary numbering of the characteristic function for a certain subset of $Fin\ n$ in this way: the i-th least significant bit of x is 1 if and only if the element i is in the subset of $Fin\ n$ encoded by x.

The membership relation over $Fin\ n$ and $Fin\ 2^n$ is now the test for the bit corresponding to the element of $Fin\ n$ in the proper position of the binary representation of the element of $Fin\ 2^n$. For an E-relation over $Fin\ n$ and $Fin\ m$, its power transpose would be the E-relation over $Fin\ n$ and $Fin\ 2^m$ which, to each element x of $Fin\ n$, assigns the element of $Fin\ 2^m$ which is the encoding of the characteristic function for the set $\{y \mid xRy\}$.

Properties of less-than

Lemma The less-than comparison between natural numbers is transitive.

Proof

$$transLt \begin{pmatrix} m \in Nat \\ n \in Nat \\ p \in Nat \end{pmatrix} \in (\mid m < n \mid) \to (\mid n < p \mid) \to \mid m < p \mid$$

$$transLt\ \mathbf{zer}\ \mathbf{zer}\ p \equiv \lambda\, h \ \to \ \mathbf{case}\ h\ \mathbf{of}$$

$$transLt\ \mathbf{zer}\ (\mathbf{suc}\, n')\ \mathbf{zer} \equiv \lambda\, h\, h' \ \to \ \mathbf{case}\ h'\ \mathbf{of}$$

$$transLt\ \mathbf{zer}\ (\mathbf{suc}\, n')\ (\mathbf{suc}\, p') \equiv \lambda\, h\, h' \ \to \ \mathbf{tt}$$

$$transLt\ (\mathbf{suc}\, m')\ \mathbf{zer}\ p \equiv \lambda\, h \ \to \ \mathbf{case}\ h\ \mathbf{of}$$

$$transLt\ (\mathbf{suc}\, m')\ (\mathbf{suc}\, n')\ \mathbf{zer} \equiv \lambda\, h\, h' \ \to \ \mathbf{case}\ h'\ \mathbf{of}$$

$$transLt\ (\mathbf{suc}\, m')\ (\mathbf{suc}\, n')\ (\mathbf{suc}\, p') \equiv transLt\ m'\ n'\ p'$$

Lemma In a less-than comparison between two natural numbers, you can substitute for them any natural numbers that are respectively equal to them.

Proof

$$substLt \begin{pmatrix} m1 \in Nat \\ m2 \in Nat \\ n1 \in Nat \\ n2 \in Nat \end{pmatrix} \in (\mid m1 == m2 \mid) \to (\mid n1 == n2 \mid) \to (\mid m1 < n1 \mid) \to \mid m2 < n2 \mid$$

$$substLt\ \mathbf{zer}\ \mathbf{zer}\ \mathbf{zer}\ n2 \equiv \lambda\, hM\, hN\, h \ \to \ \mathbf{case}\ h\ \mathbf{of}$$

$$substLt\ \mathbf{zer}\ \mathbf{zer}\ (\mathbf{suc}\, n1')\ \mathbf{zer} \equiv \lambda\, hM\, hN \ \to \ \mathbf{case}\ hN\ \mathbf{of}$$

$$substLt\ \mathbf{zer}\ \mathbf{zer}\ (\mathbf{suc}\, n1')\ (\mathbf{suc}\, n2') \equiv \lambda\, hM\, hN\, h \ \to \ \mathbf{tt}$$

$$substLt\ \mathbf{zer}\ (\mathbf{suc}\, m2')\ n1\ n2 \equiv \lambda\, hM \ \to \ \mathbf{case}\ hM\ \mathbf{of}$$

$$substLt\ (\mathbf{suc}\, m1')\ \mathbf{zer}\ n1\ n2 \equiv \lambda\, hM \ \to \ \mathbf{case}\ hM\ \mathbf{of}$$

$$substLt\ (\mathbf{suc}\, m1')\ (\mathbf{suc}\, m2')\ \mathbf{zer}\ n2 \equiv \lambda\, hM\, hN\, h \ \to \ \mathbf{case}\ h\ \mathbf{of}$$

$$substLt\ (\mathbf{suc}\, m1')\ (\mathbf{suc}\, m2')\ (\mathbf{suc}\, n1')\ \mathbf{zer} \equiv \lambda\, hM\, hN \ \to \ \mathbf{case}\ hN\ \mathbf{of}$$

$substLt\ (\mathbf{suc}\, m1')\ (\mathbf{suc}\, m2')\ (\mathbf{suc}\, n1')\ (\mathbf{suc}\, n2') \equiv substLt\ m1'\ m2'\ n1'\ n2'$

Translating binary into unary

Explanation For clarity, we introduce the name 2 for the constant value of *Nat* representing the number two.

Definition

$2\ \in Nat$

$2 \equiv \mathbf{suc}\ \ \mathbf{suc}\ (\mathbf{zero})$

A binary number will be represented as a vector of Boolean values.

Definition

$Bin\ (n\ \in\ Nat) \in Set$

$Bin\, n \equiv Vec\, Bool\, n$

Converting a binary number to a unary representation, that is passing from the *Bin* to the *Nat* type, is done by operating inside *Nat* by calculating the value of *Nat* corresponding to the binary one. An auxiliary function translates true and false to one and zero, respectively.

Definition

$boolToNat\ (x\ \in\ Bool) \in Nat$

$boolToNat\, \mathbf{true} \equiv 1$

$boolToNat\, \mathbf{false} \equiv 0$

$$binToNat \begin{pmatrix} n \in Nat \\ x \in Bin\, n \end{pmatrix} \in Nat$$

$binToNat\, \mathbf{zer}\, x \equiv \mathbf{zer}$

$binToNat\ (\mathbf{suc}\, n')\ (b\ :\ x') \equiv boolToNat\, b\ +\ (binToNat\, n'\, x'\ *\ 2)$

Auxiliary arithmetic properties

$$lemSucc \begin{pmatrix} m \in Nat \\ n \in Nat \end{pmatrix} \in (\!|\, m < n \,|\!) \to |\, \mathbf{suc}\, m < \mathbf{suc}\, n \,|$$

$lemSucc\ \mathbf{zer}\,\mathbf{zer} \equiv \lambda\, h \to$ **case** h **of**

$lemSucc\ \mathbf{zer}\ (\mathbf{suc}\, n') \equiv \lambda\, h \to \mathbf{tt}$

$lemSucc\ (\mathbf{suc}\, m')\ \mathbf{zer} \equiv \lambda\, h \to$ **case** h **of**

$lemSucc\ (\mathbf{suc}\, m')\ (\mathbf{suc}\, n') \equiv \lambda\, h \to h$

$$lemSuccSucc \begin{pmatrix} m \in Nat \\ n \in Nat \end{pmatrix} \in (\!|\, m < n \,|\!) \to |\, (2 + m) < (2 + n) \,|$$

$lemSuccSucc\ m\, n \equiv \lambda\, h \to lemSucc\ (\mathbf{suc}\, m)\ (\mathbf{suc}\, n)\ (lemSucc\ m\, n\, h)$

$$lemTwice1 \begin{pmatrix} m \in Nat \\ n \in Nat \end{pmatrix} \in (\!|\, m < n \,|\!) \to |\, (m * 2) < (n * 2) \,|$$

$lemTwice1\ \mathbf{zer}\,\mathbf{zer} \equiv \lambda\, h \to$ **case** h **of**

$lemTwice1\ \mathbf{zer}\ (\mathbf{suc}\, n') \equiv \lambda\, h \to \mathbf{tt}$

$lemTwice1\ (\mathbf{suc}\, m')\ \mathbf{zer} \equiv \lambda\, h \to$ **case** h **of**

$lemTwice1\ (\mathbf{suc}\, m')\ (\mathbf{suc}\, n') \equiv \lambda\, h \to$
$lemSuccSucc\ (\mathbf{suc}\, m' * 2)\ (\mathbf{suc}\, n' * 2)\ (lemTwice1\ m'\, n'\, h)$

$$lemTwice2 \begin{pmatrix} m \in Nat \\ n \in Nat \end{pmatrix} \in (\!|\, (m * 2) < (n * 2) \,|\!) \to |\, (\mathbf{suc}\, m * 2) < (\mathbf{suc}\, n * 2) \,|$$

$lemTwice2\ \mathbf{zer}\,\mathbf{zer} \equiv \lambda\, h \to$ **case** h **of**

$lemTwice2\ \mathbf{zer}\ (\mathbf{suc}\, n') \equiv \lambda\, h \to \mathbf{tt}$

$lemTwice2\ (\mathbf{suc}\, m')\ \mathbf{zer} \equiv \lambda\, h \to$ **case** h **of**

$lemTwice2\ (\mathbf{suc}\, m')\ (\mathbf{suc}\, n') \equiv \lambda\, h \to h$

$$lemTwice3 \begin{pmatrix} m \in Nat \\ n \in Nat \end{pmatrix} \in (\!|\, (m * 2) < (n * 2) \,|\!) \to |\, \mathbf{suc}\ (m * 2) < (n * 2) \,|$$

$lemTwice3\ \mathbf{zer}\ \mathbf{zer} \equiv \lambda\, h\ \rightarrow$ **case** h **of**

$lemTwice3\ \mathbf{zer}\ (\mathbf{suc}\, n') \equiv \lambda\, h\ \rightarrow\ \mathbf{tt}$

$lemTwice3\ (\mathbf{suc}\, m')\ \mathbf{zer} \equiv \lambda\, h\ \rightarrow$ **case** h **of**

$lemTwice3\ (\mathbf{suc}\, m')\ (\mathbf{suc}\, n') \equiv \lambda\, h\ \rightarrow$
$lemSuccSucc\ (\mathbf{suc}\ (m' * 2))\ (n' * 2)\ (lemTwice3\, m'\, n'\, h)$

$lemTwice4\ (n\ \in\ Nat) \in |\ (2\,\hat{}\,\mathbf{suc}\, n) == (2 * (2\,\hat{}\, n))\ |$

$lemTwice4\ n \equiv exp\_any\_add\ 2\ 1\ n$

$lemTwice5\ (n\ \in\ Nat) \in |\ ((2\,\hat{}\, n) * 2) == (2\,\hat{}\,\mathbf{suc}\, n)\ |$

$$lemTwice5\ n \equiv \begin{array}{l} NatSetoid.Nat.tran\ ((2\,\hat{}\, n) * 2)\ (2 * (2\,\hat{}\, n))\ (2\,\hat{}\,\mathbf{suc}\, n) \\ \quad (com\_mul\ (2\,\hat{}\, n)\ 2) \\ \quad (NatSetoid.Nat.sym\ (2\,\hat{}\,\mathbf{suc}\, n)\ (2 * (2\,\hat{}\, n))\ (lemTwice4\, n)) \end{array}$$

Auxiliary functions of finite set elements

We define a function to convert a unary number m to a finite number of the form Fin n. This of course only makes sense when $m < n$.

$$toFin \left(\begin{array}{l} m \in Nat \\ n \in Nat \\ h \in |\, m < n \,| \end{array} \right) \in Fin\, n$$

$toFin\ \mathbf{zer}\ (\mathbf{suc}\, n')\ h \equiv \mathbf{zer}$

$toFin\ (\mathbf{suc}\, m')\ (\mathbf{suc}\, n')\ h \equiv \mathbf{suc}\ (toFin\, m'\, n'\, h)$

We also define a function that produces the proof that the unary value of a *Fin n* finite number is actually less than n. The unary value is obtained with the Alfa library function *valFin*.

$$fromFinLt \left(\begin{array}{l} n \in Nat \\ x \in Fin\, n \end{array} \right) \in |\, valFin\, n\, x < n \,|$$

$fromFinLt\ (\mathbf{suc}\, n')\ \mathbf{zer} \equiv \mathbf{tt}$

$fromFinLt\ (\mathbf{suc}\, n')\ (\mathbf{suc}\, x') \equiv fromFinLt\, n'\, x'$

Power relations

A Boolean relation over finite sets can be viewed, once we fix the first argument, as a Boolean predicate over the second set. It is very easy to create a vector of the values of the predicate over the second set, that is, a binary value. The following lemma guarantees that the translation of this binary number into a unary one is within the allowed range for the finite powerset of the original second finite set.

Lemma

$$lemBinToNat \begin{pmatrix} n \in Nat \\ x \in Bin\, n \end{pmatrix} \in |\, binToNat\, n\, x < (2\, \hat{}\, n)\, |$$

$lemBinToNat\; \mathbf{zer}\; x \equiv \mathbf{tt}$

$lemBinToNat\ (\mathbf{suc}\, n')\ (\mathbf{true}\ :\ x') \equiv$
$\quad substLt\ (binToNat\ (\mathbf{suc}\, n')\ (\mathbf{true}\ :\ x'))$
$\qquad (binToNat\ (\mathbf{suc}\, n')\ (\mathbf{true}\ :\ x'))\ ((2\, \hat{}\, n') * 2)\ (2\, \hat{}\, \mathbf{suc}\, n')$
$\qquad (NatSetoid.Nat.ref\ (binToNat\ (\mathbf{suc}\, n')\ (\mathbf{true}\ :\ x')))$
$\qquad (lemTwice5\, n')\ (lemTwice3\ (binToNat\, n'\, x')\ (2\, \hat{}\, n')$
$\qquad\quad (lemTwice1\ (binToNat\, n'\, x')\ (2\, \hat{}\, n')\ (lemBinToNat\, n'\, x')))$

$lemBinToNat\ (\mathbf{suc}\, n')\ (\mathbf{false}\ :\ x') \equiv$
$\quad substLt\ (binToNat\ (\mathbf{suc}\, n')\ (\mathbf{false}\ :\ x'))$
$\qquad (binToNat\ (\mathbf{suc}\, n')\ (\mathbf{false}\ :\ x'))\ ((2\, \hat{}\, n') * 2)\ (2\, \hat{}\, \mathbf{suc}\, n')$
$\qquad (NatSetoid.Nat.ref\ (binToNat\ (\mathbf{suc}\, n')\ (\mathbf{false}\ :\ x')))$
$\qquad (lemTwice5\, n')\ (lemTwice1\ (binToNat\, n'\, x')\ (2\, \hat{}\, n')$
$\qquad\quad (lemBinToNat\, n'\, x'))$

Definition: power relation Now we can define the power relation of a Boolean relation. The definition simply asserts that two elements x and y will be in the power relation if y is equal to the finite number that results from transforming the relational image set $R(x)$ into a Boolean vector (that is, a binary number).

$powerRel\ (m, n\ \in\ Nat) \in$
$BoolHetRel\ (Fin\, m)\ (Fin\, n)\ \to\ BoolHetRel\ (Fin\, m)\ (Fin\ (2\, \hat{}\, n))$

$powerRel\, m\, n \equiv \lambda\, R\, x\, y\ \to$
$$\mathbf{let} \left[\begin{array}{l} b \in Bin\, n \\ \quad b\ \equiv\ lamVec\, Bool\, n\, \lambda\, z\ \to\ R\, x\, z \end{array} \right.$$
$\mathbf{in}\, eqFin\ (2\, \hat{}\, n)\ y\ (toFin\ (binToNat\, n\, b)\ (2\, \hat{}\, n)\ (lemBinToNat\, n\, b))$

Membership relations: unary into binary

The translation is not carried on completely, that is we don't produce a whole vector for the unary natural number. Instead we create a function that can test if any bit position of the resulting translation would be on or off. That is, we define directly the membership relation with this test function so that x has y as a member if the position corresponding to y in the binary number version of x is set on.

Definition

$$divBy2\ (n\ \in\ Nat) \in Times\ Nat\ Bool$$

$$divBy2\ \mathbf{zer} \equiv \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv \mathbf{zer} \\ snd \equiv \mathbf{false} \end{array}\right.\end{array}$$

$$divBy2\ (\mathbf{suc}\,\mathbf{zer}) \equiv \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv \mathbf{zer} \\ snd \equiv \mathbf{true} \end{array}\right.\end{array}$$

$$divBy2\ (\mathbf{suc}\ (\mathbf{suc}\,m)) \equiv \begin{array}{ll} \textbf{let} & \left[\begin{array}{l} aux \in Times\ Nat\ Bool \\ aux\ \equiv\ divBy2\ m \end{array}\right. \\ \textbf{in} & \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} fst \equiv \mathbf{suc}\, aux.fst \\ snd \equiv aux.snd \end{array}\right.\end{array} \end{array}$$

$$testBit \left(\begin{array}{l} m \in Nat \\ n \in Nat \end{array} \right) \in Bool$$

$$testBit\ \mathbf{zer}\ n \equiv (divBy2\ n)\ .snd$$
$$testBit\ (\mathbf{suc}\ m')\ n \equiv testBit\ m'\ (divBy2\ n)\ .fst$$

Definition: membership

$$membRel\ (n\ \in\ Nat) \in BoolHetRel\ (Fin\ (2\ \hat{}\ n))\ (Fin\ n)$$

$$membRel\ n \equiv \lambda\ x\ y\ \rightarrow\ testBit\ (valFin\ n\ y)\ (valFin\ (2\ \hat{}\ n)\ x)$$

We have now shown all the pieces necessary to show that finite decidable relations form a Boolean allegory. There are some (probably long) very tedious proofs necessary for satisfying the formal signature, but we feel that it should be clear the pieces do satisfy the corresponding properties. A complete instantiation of the Boolean allegory signature would be a matter for future work.

Chapter 5

Relational Programs

In their book *Algebra of Programming* [10], Bird and de Moor pursue a relational approach to programming. Their motivation for the use of relation calculus is that it gives a freedom in specification and proof that would not be achieved by the use of a calculus of functions alone. The idea is to go through some program calculation, and arrive at a relation defined by a recursive equation. Finally, this equation is refined to a recursive program that delivers a function, and the result is translated into a functional programming language (for example, basic Haskell).

Their work is written in the usual informal mathematical style, but the desire for mechanization is clear. A natural question is what the setting for such an effort would be. Ideally, the formalization should accommodate both a calculus of relations and a functional programming language. It is of course possible either to implement a specific logic capturing the relation calculus used by Bird and de Moor directly or via a translation into some standard general-purpose logic such as ZF set theory or higher order logic (both of which have computer systems supporting them).

Here we suggest a third alternative, namely using Martin-Löf's type theory. This theory naturally accommodates a calculus of relations, as well as a calculus of functions, as we hope the preceding chapters have shown. In this setting, the notion of a computable function is a primitive one. We will begin by discussing how one can represent partial functions in type theory, its connection to representing relations. Then we move to discussing some categorical concepts for dealing with data types in relational settings, and how they get represented in our type-theoretic version of allegories.

5.1 Functions and relations in MLTT

Martin-Löf's type theory needs a calculus of relations too for an important reason: functions in this theory are total and one way to represent partial functions is as univalent relations. The concept of partial function is not a primitive one, and there are different ways one can represent them in type theory. The first approach is based on representing partial recursive functions as relations.

Consider for example the minimization operator

$$\text{mu} : (N \rightharpoonup N) \rightharpoonup N$$

(here $N \rightharpoonup N$ denotes the set of partial functions on N) for which the specification is

$$\text{mu}\, f = n \quad \text{iff} \quad f\, n = 0 \,\&\, \forall n' < n. f\, n' \neq 0$$

A recursive definition in a functional programming language is as follows:

$$\begin{array}{rcl} \text{mu}\ f & = & \text{if}\ f\,0 == 0 \quad \text{then}\ 0 \\ & & \qquad \text{else}\ s(\text{mu}\ \ (f \circ s)) \end{array}$$

where s is the successor function for natural numbers, == is the equality comparison for them, <> the corresponding inequality, and $\circ$ denotes the usual function composition operation. Note that this is a partial function, since mu f is undefined if $f\,n$ <> 0 for all n.

How can we represent mu in Martin-Löf's type theory? One possibility is as the relation (now $N \to N$ denotes the set of total functions again):

$$\mu : (N \to N) \to N \to Set$$

such that mu $f = n$ iff $\mu\, f\, n$, inductively defined by the rules:

$$\frac{c :\ T(f\,0 == 0)}{\mu_0\, c :\ \mu f\, 0}$$

$$\frac{c :\ T(f\,0 <> 0) \qquad p :\ \mu\,(f \circ s)\, n}{\mu_1\, c\, p :\ \mu\, f\,(s\, n)}$$

where ==: $N \to N \to Bool$, <> is its Boolean negation, and T raises Boolean values to sets (the empty set for false and the singleton set for true).

Actually, we can and prefer to have the following decidable relation:

$$\mu : (N \to N) \to N \to Bool$$

with definition:

$$\begin{array}{rcl} \mu\, f\, 0 & = & (f\, 0\ ==\ 0) \\ \mu\, f\,(s\, n) & = & (f\, 0 <> 0) \,\&\, \mu\,(f \circ s)\, n \end{array}$$

There are two points to make about this approach, though. First, for the relation to actually define a partial function, we need to prove that it is univocal (i.e., at most a single result for each value in its domain). Second, there appears to be no computational content, in the sense that μ doesn't tell us what the value of $\mu\, f$ is (if there is one), only whether $\mu\, f == n$ given n. To the first objection, we can reply that it is possible to prove in type theory that

$$\mu\, f\, n \ \&\ \mu\, f\, n' \ \Rightarrow\ T(n == n')$$

To the second objection, we can say that although there is no computational content, we can still reason about it, and do so quite nicely given the power of the calculus of relations.

In a method proposed by Bove & Capretta [11], partial recursive functions can be coded as total functions from their original domain along with a domain predicate. In Bove's approach one systematically generates an inductive definition of the domain predicate from the recursion equations of a partial recursive function. More generally, we can represent a relation using a domain predicate too. In general, given a relation

$$R : A \to B \to Set$$

we can define its domain as

$$D : A \to Set$$

$$D\, a = \exists b : B.a\, R\, b$$

Then we can define a function

$$f : (a : A) \to D\, a \to B$$

$$f\, a\, (b, p) = b$$

and if R is univocal, $f\, a\, (b, p)$ is uniquely determined by a.

In the converse direction, given a function

$$f : (a : A) \to D\, a \to B$$

we can define a relation

$$R : A \to B \to Set$$

by

$$a\, R\, b \quad \text{iff} \quad \exists p : D\, a. f\, a\, p = b$$

But note that such a function f only represents a good partial function if and only if its value is independent of the proof $p : D\, a$. That is, if $f\, a\, p = f\, a\, p'$ for all $p, p' : D\, a$. (Note that this requires B to have an equality over it, then.)

5.2 A translation method

Moreover, just as there is a systematic way (by the method of Bove and Capretta, and under certain restrictions) to translate the definition of a general recursive function $f : A \to B$ into a termination predicate D and a total function $f' : (a : A) \to D\,a \to B$, there is also a systematic way of translating such a function definition into an inductively defined relation. For this, first consider the primitive recursive functions where $A = N^k$ and $B = N$.. Each such function is directly represented in type theory. Now consider a Kleene partial recursive function, $f : N^k \to N$. It is translated into a relation

$$f' : N^k \to N \to Set$$

(*Bool* instead of *Set* if we want decidable relations only, and the situation is such that this is possible).

If $f : N^k \to N \to N$ is primitive recursive, then the partial recursive function $\mu\, f : N^k \to N$ is given by the recursion equation

$$\begin{array}{rcl} \mu\, f(n_1, \ldots, n_k) & = & \text{if } f(n_1, \ldots, n_k)0 == 0 \text{ then } 0 \\ & & \quad \text{else } s(\mu(\lambda x. f(n_1, \ldots, n_k)(s\,x))) \end{array}$$

It is represented by the relation $\mu\, f : N^k \to N \to Bool$, where

$$\begin{array}{rcl} \mu\, f(n_1, \ldots, n_k)0 & = & f(n_1, \ldots, n_k) == 0 \\ \mu\, f(n_1, \ldots, n_k)(s\,n) & = & \mu\,(\lambda x. f(n_1, \ldots, n_k)(s\,x))(n_1, \ldots, n_k)n \end{array}$$

To make matters even more precise, consider the representation according to Bove's method, for $k = 0$, of μ as above. Introducing the domain predicate, it becomes $\mu' : (f : N \to N) \to D\, f \to N$, where D is given by the rules:

$$\frac{p : f\,0 == 0}{c_0\, f\, p : D\, f}$$

$$\frac{p : f\,0 <> 0 \qquad q : D\,(f \circ s)}{c_1\, q : D\, f}$$

and the program definition is pretty much the same as in the previous section:

$$\begin{array}{rcl} \mu'\, f\,(c_0\, f\, p) & = & 0 \\ \mu'\, f\,(c_1\, f\, p\, q) & = & s(\mu'\,(f \circ s)\, q) \end{array}$$

We omit the relational representation, since the definition is exactly the same as in that section.

5.3 An alternative translation

The following method only works for functions with a decidable termination predicate. (We remark that this method is used for representing some Haskell programs in Agda. Cf. recent work by the members of the Cover Project [72]).

Another possible representation of a partial function $f : A \to B$ is by using the *Maybe* type:

$$Maybe : Set \to Set$$

$$Maybe\ X \;=\; \mathbf{data}\ Just\ (x : X) \mid Nothing$$

The type of the new representation is now $f' : A \to Maybe\ B$, such that:

$$\begin{array}{lcl} f'\,a & = & Just\ b\ \text{ iff }\ f\,a == b \\ f'\,a & = & Nothing\ \text{ iff }\ f\,a\ \text{ is undefined} \end{array}$$

Note, however, that this requires that it is decidable whether $f\,a$ is defined or not.

Given such an $f' : A \to Maybe\ B$, we can get a relation $R : A \to B \to Bool$ by

$$a\,R\,b \quad \text{iff} \quad f\,a == Just\ b$$

where here $==: Maybe\ B \to Maybe\ B \to Bool$.

To see the connection with the way of expressing partial functions using a domain predicate, consider $f : (a : A) \to |D\,a| \to Bool$, with D decidable and Boolean-valued. We can then get $f' : A \to Maybe\ B$ by the definition

$$\begin{array}{lcl} f'\,a & = & \text{if } D\,a \text{ then } f\,a\,\mathbf{tt} \\ & & \quad \text{else } Nothing \end{array}$$

(where **tt** is the only element of the data type for the singleton set, which represent the true propositions).

5.4 A simple example

To illustrate matters regarding the connection between functions and relations in type theory, we will go through an example involving natural number division. This operation is a partial function $div : N^2 \to N$, in that the second component of the argument pair cannot be zero. A recursive definition looks like this:

$$\begin{array}{lcl} \text{div}\ (m,n) & = & \text{if } m < n \\ & & \quad \text{then } 0 \\ & & \quad \text{else } s(\text{div}(m-n,n)) \end{array}$$

According to the method by Bove, we represent it with a domain predicate $D : N^2 \to Bool$:

$$div' : (x : N^2) \to D\,x \to N$$

where D is defined inductively by the following rule:

$$\frac{n : N \qquad 0 < n}{c\,n : D(m, n)}$$

(Here it is easy to see that the domain predicate is decidable, so we can alternatively define $div : (x : N) \to Maybe\ N$.)

We can also view it as a relation:

$$div_R : N^2 \to N \to Set$$

whose specification would be

$$(m, n)\ div_R\, q \ \text{ iff } \ \exists p : D(m, n).div'(m, n)p == q$$

Giving the relation in the shape of inductive rules we get

$$\frac{m < n}{(m, n)\ div_R\ 0}$$

$$\frac{m \geq n \qquad n > 0 \qquad (m - n, n)\ div_R\ k}{(m, n)\ div_R\ s(k)}$$

In fact, this is naturally expressed as a logic program too:

$$\begin{array}{rcl} \mathrm{Div}\ m\,n\,0 & \text{:-} & m < n \\ \mathrm{Div}\ m\,n\,(s\,k) & \text{:-} & \mathrm{Div}\ (m - n)\,n\,k \end{array}$$

5.5 Relational semantics

An interesting question at this point in the exploration of the topic of relational programs is: if the language of Kleene partial recursive equations is evaluated using the lazy evaluation strategy, what would its relational semantics be? Here we mean having lazy natural numbers, that is, a computation of a natural number would terminate as soon as we reach a weak head normal form 0 or $s(a)$ irrespective of a. Then a would be computed lazily in the same way. An answer would have to give a semantics for the whole of the language of Kleene partial recursive functions. Here we present an application of some ideas of Scott [63] and Martin-Löf dating back to the 1980s. The reason they are worth mentioning in the context of our current discussion on relational programs is that Bird and de Moor work with a lazy functional language to implement such programs.

In the standard call-by-value strategy, relational semantics captures it nicely by saying that a function $f : N \to N$ is given its semantics by a relation $F \subseteq N \times N$.Then, when one says that mFn, this means that if the value of the input is m, then the value of the output is n. If we wished to have lazy evaluation, we cannot capture the fact that if f is a constant function $f\,x = n$ for some $n \in N$, then $f \bot = n$, i.e. that f returns n even if its argument does not terminate.

To give a relational semantics for such a lazy system, we can use some ideas from domain theory: lazy natural numbers and approximable mappings.

A lazy natural number is either undefined (denoted by $\bot$, or sometimes Δ; we will prefer the latter), or 0, or $s(u)$ where u is a lazy natural number. If we were to draw a Hasse diagram with Δ at the lowest position, then Δ is covered by 0 and $s(\Delta)$, 0 is not covered by anything, $s(\Delta)$ is covered by $s(0)$ and $s(s(\Delta))$, $s(0)$ is covered by nothing, and so on.

An approximable mapping (on $N \to N$) is a relation $F \subseteq N \times N$ with some special properties (which we don't get into for the purposes of this discussion, see [63] for details). The idea is that the lazy program $f : N \to N$ is modeled by the approximable mapping $F \subseteq N \times N$ such that uFv means: if you get u as an approximation of the input, then you get v as an approximation of the output. We can say that uFv iff you get at least v by computing u.

Such an approximable mapping has the properties:

- $uF\Delta$
- if uFv and $v' \leq v$, then uFv'

where $\leq$ means the standard ordering on the domain of lazy natural numbers. (Again, other properties might hold in more complex domains; we don't get into the details here).

Let's consider some common functions, such as $K_0 : N \to N$, the function which always returns 0. It is modeled by the approximable mapping

$$uF_0v \quad \text{iff} \quad v = \Delta \text{ or } v = 0 \quad (\text{for all } u)$$

We could also write $u[\![0]\!]v$ iff $v \leq 0$, using the notation $[\![\cdot]\!]$ for the approximable mapping.

The meaning of the successor function $s : N \to N$ is

$$uF_sv \quad \text{iff} \quad v \leq s(u)$$

Again, this could also be written as $u[\![s]\!]v$ iff $v \leq s(u)$.

The meaning of the composition of two functions is just relational composition:

$$[\![g \circ f]\!] = [\![g]\!] \cdot [\![f]\!]$$

(please note that $\cdot$ is, as in previous chapters, relational composition).

Consider now the μ-operator. Let $f : N \to N$ be modeled by the approximable mapping $[\![f]\!]$. We have $\mu f : N$, so $[\![\mu f]\!] \subseteq N$. Expressed as rules:

$$\frac{0[\![f]\!]0}{0 \in [\![\mu f]\!]}$$

$$\frac{u \in [\![\mu(f \circ s)]\!] \quad 0[\![f]\!]s(\Delta)}{s(u) \in [\![\mu f]\!]}$$

This means we need to add that elements $n : N$ are modeled by downward closed subsets $[\![n]\!]$ of the set of lazy natural numbers.

In this way, we can go on and give a lazy semantics for the whole of the language of Kleene partial recursive functions. We will not here go further into the discussion of how to combine relational programming and lazy evaluation.

5.6 E-functors in LT-allegories

In the work of Bird and de Moor, the so-called polynomial functors play an important role in the category of relations. Recursive datatypes are modeled by initial algebras of such functors. Their class is defined inductively as that consisting of the identity functor, all constant functors, and the composition, pointwise sum and pointwise product of polynomial functors.

In our setting of E-categories (generated from LT-categories), the corresponding notion to that of a functor would be an E-functor. The definition follows the standard formulation, except that the equalities mentioned in it are those of the corresponding E-relation types. Hence an E-functor F between two E-categories C and C' has a component that applies to objects:

$$F_{\text{obj}} : C_{\text{obj}} \to C'_{\text{obj}}$$

and a component that applies to arrows (in this case, relations):

$$\text{for } R : A \to B \to \mathit{Set}, \;\; F_{\text{arr}}\, R : F_{\text{obj}}\, A \to F_{\text{obj}}\, B \to \mathit{Set}$$

It has to verify the usual properties:

$$F(\text{id}_A) = \text{id}_{FA}$$

(this equality is the equality relation on *HetRel A B*, that is for the type of relations $A \to A \to \mathit{Set}$)

$$F(R \cdot S) = F\,R \cdot F\,S$$

(this equality is the one for relations $A \to C \to Set$, assuming $R : A \to B \to Set$ and $S : B \to C \to Set$) and also preserve equality of arrows, that is

$$R = S \implies F\,R = F\,S$$

It is helpful to remark that this means the object part of an E-functor is simply a functor, while the arrow part is a setoid map, i.e. a function between the underlying sets of the setoid parameters which preserves setoid equality. Furthermore the identity and composition are preserved with respect to the setoid equality.

Now we briefly show that the polynomial functors are indeed E-functors for the LT-category of setoids and E-relations. For this, the actual proofs are of two E-relations being equal, which is in turn proved by inclusion of each the two E-relations in the other.

- the identity functor leaves the setoids and E-relations unchanged. It clearly verifies the two properties for E-functor.

- consider a constant functor K_A, which maps every object to A and every relation to the identity relation id_A. It clearly verifies the preservation of identities. For the preservation of composition, the composition on the right hand side is of two identities, so it's equal to the same identity (all are over the setoid A).

- for the composition of two polynomial E-functors, it clearly verifies the preservation of identities. The preservation of composition follows by applying the preservation of each of the composed functors in turn.

- for the pointwise sum functor $F + G$, represented with the help of the Alfa datatype
  $$Sum\;X\,Y = \textbf{data}\;Inl\;(x : X)\,|\,Inr\;(y : Y)$$
  such that for objects the definition of $F + G$ is by case analysis of its argument $(A : Maybe\;\mathrm{C}_{\mathrm{obj}}C'_{\mathrm{obj}})$, and for an arrow $(F + G)h$ is defined by case analysis of the argument of the resulting function $Fh + Gh$: identity preservation follows by simplification of the case analysis.

- for the pointwise product function $F \times G$, defined by a standard pairing construction: identity preservation clearly holds; preservation of compositions follows from the absorption law for products.

5.7 Recursion and relations

When dealing with datatypes categorically, a fundamental notion is that of a catamorphism. This is a powerful tool for program specification and programming, since it captures the process of performing structural recursion through a value of an algebraic datatype: considering the datatype as an initial algebra, its catamorphisms are the unique homomorphisms between its initial algebra and an arbitrary algebra of the same datatype. The theory of allegories works with relational versions of catamorphisms, and we will now go through a simple example both to illustrate the idea and at the same time exploring how it is represented in our type-theoretic setting.

For F an E-functor, an F-algebra is an arrow α of type $\mathrm{Hom}(F\,A, A)$, and A is called the carrier of the algebra. Initial algebras are denoted as follows:

$$F\,A \xrightarrow{\alpha} A$$

where the A is an object representing the terms of such an initial structure. For the polynomial functors (products, coproducts, identities and constants) these initial algebras always exist in the category of sets and functions. For any F-algebra f, its corresponding catamorphism is denoted $(\!|f|\!)$. It satisfies:

$$\begin{array}{ccc} F\,T & \xrightarrow{\alpha} & T \\ \downarrow{\scriptstyle F(\!|f|\!)} & & \downarrow{\scriptstyle (\!|f|\!)} \\ F\,A & \xrightarrow{f} & A \end{array}$$

As our running example, consider natural numbers as a data type (defined as in Agda/Alfa notation):

$$\textbf{data}\ \mathit{Nat}\ \equiv \mathit{zero} \mid \mathit{succ}\ \mathit{Nat}$$

We will later use lists in Alfa for examples and proofs.

Now $\alpha = [\mathit{zero}, \mathit{succ}] : F\,1{+}\mathit{Nat} \to \mathit{Nat}$ is the initial algebra for the functor $F\,A = 1{+}A$, so we have $F\,h = id_1 + h$. Here we use zero for the arrow $1 \to \mathit{Nat}$, and $[f, g] : A + B \to C$ denotes the mediating isomorphism for the coproduct of $f : A \to C$ and $g : B \to C$).

Now assume we have a relation for the same type of algebras:

$$1 + C \xrightarrow{[R_0, R_s]} C$$

where the $R_0 : 1 \to C \to \mathit{Set}$ and $R_s : C \to C \to \mathit{Set}$ so that $R = [R_0, R_s]$. We want to obtain the kind of relational catamorphism that is provided in the original algebra of allegories, that is, one that satisfies:

$$\begin{array}{ccc} 1 + N & \xrightarrow{[\mathit{zero}, \mathit{succ}]} & N \\ \downarrow{\scriptstyle 1+(\!|R|\!)} & & \downarrow{\scriptstyle (\!|R|\!)} \\ 1 + C & \xrightarrow{R} & C \end{array}$$

The definition of the relational catamorphism $(\!|R|\!) : N \to C \to Set$ is as follows:

$$\begin{array}{rcl} 0\,(\!|R|\!)\,c & \text{iff} & \mathbf{tt}\,R_0\,c \\ (s\,n)\,(\!|R|\!)\,c & \text{iff} & \exists c' : C.\; n\,(\!|R|\!)\,c' \;\&\; c'\,R_s\,c \end{array}$$

where **tt** is the only element of the singleton type, which represents the true propositions by the Curry-Howard isomorphism. Note that this is a primitive recursive definition of a family $C \to Set$, i.e., it is both a higher-type recursion and also a set-valued one. It is easy to check that this definition makes the initial algebra diagram commute (up to equality of relations).

Let us now consider the E-category of E-relations. We need to check that $(\!|R|\!)$ is an E-relation, i.e. that the following holds: if $n =_N n'$ and $c =_C c'$, then $n(\!|R|\!)c$ implies $n'(\!|R|\!)c'$. This can be proved by induction on n, using that R_0 and R_s are E-relations.

Now we need to show that in type theory, initiality also holds, and $(\!|R|\!)$ is the unique mediating arrow in the LT-allegory (and E-category) of E-relations. Suppose there exists another relation $Q : N \to C \to Set$ that also satisfies the preceding commutative diagram. That is,

$$x\,([zero, succ] \cdot Q)\,c \;\text{ iff }\; x\,((1+Q)\cdot R)\,c$$

By induction on the first argument n of Q (and R), consider first $n = 0$:

$$\begin{array}{rcl} 0\,(\!|R|\!)\,c & \text{iff} & \mathbf{tt}\,R_0\,c \\ & \text{iff} & (Inl\,\mathbf{tt})\,R\,c \\ & \text{iff} & (Inl\,\mathbf{tt})\,((1+Q)\cdot R)\,c \\ & \text{iff} & (Inl\,\mathbf{tt})\,(\alpha\cdot Q)\,c \\ & \text{iff} & 0\,Q\,c \end{array}$$

since it must be the case that the bridging element in the last composition is 0 (it is the only one related to *Inl* **tt** by α, since we have $0 = Inl\,\mathbf{tt}$).

Now consider the case $n = s\,n'$:

$$\begin{array}{rcl} (s\,n')\,(\!|R|\!)\,c & \text{iff} & \exists c' : C.\; n'\,(\!|R|\!)\,c' \;\&\; c'\,R_s\,c \\ & \text{iff} & \exists c' : C.\; n'\,Q\,c' \;\;\&\;\; c'\,R_s\,c \;\; \text{(by ind. hyp.)} \\ & \text{iff} & \exists c' : C.\; (Inr\,n')\,(1+Q)\,(Inr\,c') \;\;\&\;\; (Inr\,c')\,R\,c \\ & \text{iff} & (Inr\,n')\,((1+Q)\cdot R)\,c \\ & \text{iff} & (Inr\,n')\,(\alpha\cdot Q)\,c \\ & \text{iff} & (s\,n')Q\,c \end{array}$$

since in the last composition α can only take $Inr\,n'$ to $s\,n'$. Hence, $(\!|R|\!)$ in this case is a proper catamorphism in the type theory version of allegories.

5.8 Additional E-relations for dealing with catamorphisms

As can be seen from the preceding section, we need some additional E-relations that we haven't introduced yet for expressing relational catamorphisms. We do so in this section, and prove that they are indeed E-relations. The definitions in this section will be carried out formally in Alfa, since we intend to use them later on within examples of the use of relational catamorphisms in simple programs, which are themselves formally defined in Alfa.

5.8.1 Product of two relations

We can view this E-relation intuitively as combining two E-relations in parallel. The product relation relates pairs of elements coming from the product of the original domains with pairs of elements coming from the product of the original codomains. Notice that a proof that this product respects equality must be provided, as we do in the second component of the structure definition.

$\mathit{prodERel}\ (A, A', B, B' \in \mathit{Setoid}, R \in \mathit{ERelation}\ A\ B, S \in \mathit{ERelation}\ A'\ B') \in$
$\mathit{ERelation}\ (A \times A')\ (B \times B')$

$$
\begin{array}{l}
\mathit{prodERel}\ A\ A'\ B\ B'\ R\ S \equiv \\
\quad \textbf{struct} \\
\quad \left[\begin{array}{l}
\mathit{rel} \equiv \lambda\, x\, y\ \rightarrow\ R.\mathit{rel}\ x.\mathit{fst}\ y.\mathit{fst}\ \&\ S.\mathit{rel}\ x.\mathit{snd}\ y.\mathit{snd} \\
\mathit{resp} \equiv \lambda\, x1\, x2\, y1\, y2\, hX\, hY\, h\ \rightarrow\
\begin{array}{l}
\textbf{struct} \\
\left[\begin{array}{ll}
\mathit{fst} \equiv & \begin{array}{l} R.\mathit{resp} \\ \ x1.\mathit{fst} \\ \ x2.\mathit{fst} \\ \ y1.\mathit{fst} \\ \ y2.\mathit{fst} \\ \ hX.\mathit{fst} \\ \ hY.\mathit{fst} \\ \ h.\mathit{fst} \end{array} \\
\mathit{snd} \equiv & \begin{array}{l} S.\mathit{resp} \\ \ x1.\mathit{snd} \\ \ x2.\mathit{snd} \\ \ y1.\mathit{snd} \\ \ y2.\mathit{snd} \\ \ hX.\mathit{snd} \\ \ hY.\mathit{snd} \\ \ h.\mathit{snd} \end{array}
\end{array} \right.
\end{array}
\end{array} \right.
\end{array}
$$

5.8.2 Relational projection

We also need a relation version of projecting from a domain that is a product into one of its components (a right one here). Once more a proof that the relation respects equality is provided, and so we have an E-relation. The relation simply picks the corresponding component from the domain argument pair.

$$outrRel\ (A, B\ \in\ Setoid) \in ERelation\ (A\ \times\ B)\ B$$

$$outrRel\ A\ B \equiv \begin{array}{l} \textbf{struct} \\ \left[\ rel \equiv \lambda\, x\, b\ \rightarrow == \ B\, x.snd\, b \right. \\ \left[\ resp \equiv \lambda\, x\, x'\, b1\, b2\, hX\, hB\, h\ \rightarrow \begin{array}{l} B.tran \\ \quad x'.snd \\ \quad b1 \\ \quad b2 \\ \quad \left(\begin{array}{l} B.tran \\ \quad x'.snd \\ \quad x.snd \\ \quad b1 \\ \quad (B.sym\, x.snd\, x'.snd\, hX.snd) \\ \quad h \end{array} \right) \\ \quad hB \end{array} \right. \end{array}$$

5.8.3 Relational case-of

The last E-relation we need is one that takes as domain a sum type, and according to the kind of value it receives as argument, it relates it through either of two relations that act on only one of the components of the sum. We call it a "case of sum" E-relation then, since it amounts to doing a case analysis on the first argument and then relating it through the appropiate relation from the two available.

$$caseSumERel\ (A, B, C\ \in\ Setoid, R\ \in\ ERelation\, A\, C, S\ \in\ ERelation\, B\, C) \in$$
$$ERelation\ (A\ +\ B)\ C$$

$$caseSumERel\, A\, B\, C\, R\, S \equiv$$

```
struct
  [                      case x of
  [ rel ≡ λ x c  →         inl a  →  R.rel a c
  [                        inr b  →  S.rel b c
  [ resp ≡ λ x1 x2 c1 c2 hX hC h  →
  [       case x1 of
  [                        case x2 of
  [         inl a1  →        inl a2  →  R.resp a1 a2 c1 c2 hX hC h
  [                          inr b2  →  elimAbsurd (rel (inr b2) c2) hX
  [                        case x2 of
  [         inr b1  →        inl a2  →  elimAbsurd (rel (inl a2) c2) hX
  [                          inr b2  →  S.resp b1 b2 c1 c2 hX hC h
```

5.9 Catamorphisms on lists

We will now formalize in detail the definition of catamorphism for lists over a setoid *A*.

5.9.1 Nil selection

We start by defining an E-relation between the singleton setoid and the setoid of lists that holds only for empty lists. That is, an R such that $R : 1 \leftarrow List\ A$ satisfying **tt** $R\,c$ iff $c = []$. As a matter of notation, we remark that {!} is the Alfa library name for the singleton setoid.

eRelIsNil (*A* ∈ *Setoid*) ∈ *ERelation* {!} (*List A*)

```
                 struct
                   [                     case l of
                   [ rel ≡ λ x l  →        []  →  T
                   [                       a : as  →  ⊥
                   [                                      case l1 of
eRelIsNil A ≡      [                                             case l2 of
                   [                                      []  →    []  →  tt
                   [ resp ≡ λ a1 a2 l1 l2 hA hL h  →               a : as  →  hL
                   [                                                case l2 of
                   [                                      a : as  →  []  →  tt
                   [                                                  a′ : as′  →  h
```

5.9.2 Cons selection

Next we need to define an E-relation that holds only between pairs (a, as) and lists l if and only if $a : as = l$. As usual, since we want to define an E-relation we have to prove that this relation respects equality.

$eRelIsCons\ (A\ \in\ Setoid) \in ERelation\ (A\ \times\ List\ A)\ (List\ A)$

$eRelIsCons\ A \equiv$
struct

$$\left[\begin{array}{l} rel \equiv \lambda\, x\, l\ \to \begin{array}{l} \textbf{case}\ l\ \textbf{of} \\ \quad [\,] \to \bot \\ \quad a\ :\ as\ \to == A\ x.fst\, a\ \&\ ==\ (List\ A)\ x.snd\, as \end{array} \end{array}\right]$$

$$\left[\begin{array}{l} resp \equiv \lambda\, x1\, x2\, l1\, l2\, hX\, hL\, h\ \to \\ \quad \textbf{case}\ l1\ \textbf{of} \\ \quad\quad [\,] \to \begin{array}{l} \textbf{case}\ l2\ \textbf{of} \\ \quad [\,] \to h \\ \quad a\ :\ as\ \to elimAbsurd\ (rel\, x2\ (a\ :\ as))\ hL \end{array} \\ \quad\quad a\ :\ as\ \to \\ \quad\quad\quad \textbf{case}\ l2\ \textbf{of} \\ \quad\quad\quad\quad [\,] \to elimAbsurd\ (rel\, x2\, [\,])\ hL \\ \quad\quad\quad\quad a'\ :\ as'\ \to \\ \quad\quad\quad\quad\quad \textbf{struct} \\ \quad\quad\quad\quad\quad \left[fst \equiv \begin{array}{l} A.tran \\ \ x2.fst \\ \ a \\ \ a' \\ \ \left(\begin{array}{l} A.tran \\ \ x2.fst \\ \ x1.fst \\ \ a \\ \ (A.sym\, x1.fst\, x2.fst\, hX.fst) \\ \ h.fst \end{array} \right) \\ \ hL.fst \end{array} \right] \\ \quad\quad\quad\quad\quad \left[snd \equiv \begin{array}{l} (List\ A)\ .tran \\ \ x2.snd \\ \ as \\ \ as' \\ \ \left(\begin{array}{l} (List\ A)\ .tran \\ \ x2.snd \\ \ x1.snd \\ \ as \\ \ ((List\ A)\ .sym\, x1.snd\, x2.snd\, hX.snd) \\ \ h.snd \end{array} \right) \\ \ hL.snd \end{array} \right] \end{array}\right]$$

5.10 Simpler catamorphism for lists

In the section immediately following this one, we construct the relational catamorphism in the category of E-relations, including proof objects. If we only wish to construct the underlying relation of the E-relation we do as follows.

$$\mathit{cataList} \begin{pmatrix} A \in \mathit{Set} \\ B \in \mathit{Set} \\ \mathit{nilR} \in \mathit{HetRel}\ \mathit{SET.Unit}\ B \\ \mathit{consR} \in \mathit{HetRel}\ (\mathit{SET.Times}\ A\ B)\ B \end{pmatrix} \in \mathit{HetRel}\ (\mathit{SET.List}\ A)\ B$$

$\mathit{cataList}\ A\ B\ \mathit{nilR}\ \mathit{consR} \equiv \lambda\, l\, b\ \rightarrow$
case l **of**
$[]\ \rightarrow\ \mathit{nilR}\ \mathbf{tt}\ b$
$a\ :\ as\ \rightarrow\ \exists\, b' \in B\,.\ \mathit{cataList}\ A\ B\ \mathit{nilR}\ \mathit{consR}\ as\ b'\ \&\ \mathit{consR}\ ((a\,,\,b'))\ b$

5.11 Catamorphisms for lists

We can now, with the auxiliary selection relations presented before, plus the extra E-relations of the previous section, give a complete definition of what a catamorphisms for lists is, and also prove that it constitutes an E-relation. Notice that the definition of the relation component of this E-relation is essentially the same as the one of the preceding subsection, with only some adjustments to allow for the E-relations that are now parameters of the definition. Once again, the proof that the relation component is an E-relation forms a necessary part of the whole definition and is provided.

$$\mathit{cataListERel} \begin{pmatrix} A \in \mathit{Setoid} \\ B \in \mathit{Setoid} \\ \mathit{nilR} \in \mathit{ERelation}\ \{!\}\ B \\ \mathit{consR} \in \mathit{ERelation}\ (A \times B)\ B \end{pmatrix} \in \mathit{ERelation}\ (\mathit{List}\ A)\ B$$

$\mathit{cataListERel}\ A\ B\ \mathit{nilR}\ \mathit{consR} \equiv$

```
struct
  [                        case l of
  [ rel ≡ λ l b →            [] → nilR.rel tt b
  [                          a : as → ∃b′ ∈ |B| . rel as b′ & consR.rel ((a , b′)) b
  [ resp ≡ λ l1 l2 b1 b2 hL hB h →
  [     case l1 of
  [                 case l2 of
  [       [] →        [] → nilR.resp tt tt b1 b2 tt hB h
  [                   x : xs →  case hL of
  [
  [       x : xs →
  [         case l2 of
  [           [] →  case hL of
  [
  [           x′ : xs′ →
  [               struct
  [                 [ fst ≡ h.fst
  [                 [ snd ≡
  [                 [       struct
  [                 [         [ fst ≡ resp xs xs′ h.fst h.fst hL.snd (B.ref h.fst) h.snd.fst
  [                 [         [ snd ≡ consR.resp ( x , h.fst ) ( x′ , h.fst ) b1 b2
  [                 [         [      ( hL.fst , B.ref ( x , h.fst ) .snd ) hB h.snd.snd
```

5.12 An example: two definitions of subsequence

We can define a relation that holds between two lists if and only if for the first one is a subsequence of the second one. This is an example of relational catamorphism taken from the book by Bird and de Moor [10], and will allow us to see how our formalization can deal with relational programming and how it corresponds to type-theoretical program constructions.

We will now give three definitions of the subsequence relation and prove their equivalence. The first one is a poinwise one which is natural to write in type theory. The second one and third one are from the book by Bird and de Moor. One using a relational catamorphism, and another corresponding to a functional program which maps a sequence to the list of its subsequences.

To begin with, we define subsequence as a relation in our type-theoretic framwork. The definition we provide is intuitive and clear, which is nice.

$subseqOf\ (A\ \in\ Setoid) \in HomRel\ (|\,List\ A\,|)$

$subseqOf\ A \equiv \lambda\, l\, m\ \rightarrow$

```
case l of
  []     → T
                 case m of
  x : xs →         []       → ⊥
                   x′ : xs′ → (== A x x′ & subseqOf A xs xs′) ∨ subseqOf A l xs′
```

Now we prove that this "subsequence of" relation is an E-relation. The proof proceeds by induction on the argument lists, and is quite straightforward.

subseqOfERel (A ∈ *Setoid*) ∈ *ERelation* (*List A*) (*List A*)

subseqOfERel A ≡

```
struct
  [ rel ≡ subseqOf A
  [ resp ≡ λ l1 l2 m1 m2 hL hM h →
      case l1 of
        [] → case l2 of { [] → tt
                          x : xs → case hL of {} }
        x : xs →
            case l2 of
              [] → case hL of {}
              x′ : xs′ →
                  case m1 of
                    [] → case h of {}
                    x0 : xs0 →
                        case m2 of
                          [] → case hM of {}
                          x1 : xs1 →
                              case h of
                                inl hdEq → inl
                                  struct
                                    [        A.tran x′ x x1
                                    [ fst ≡   (A.sym x x′ hL.fst)
                                    [         (A.tran x x0 x1 hdEq.fst hM.fst)
                                    [ snd ≡ resp xs xs′ xs0 xs1
                                    [         hL.snd hM.snd hdEq.snd
                                inr hdNeq → inr (resp (x : xs) (x′ : xs′) xs0 xs1
                                                  hL hM.snd hdNeq)
```

It is interesting to compare this with how we would define the same subsequence E-relation between two lists using relational catamorphisms, using the definition of catamorphisms over lists in type theory that we provided before in this chapter. We show now the definition in this style.

$$subseq' \ (A \ \in \ Setoid) \in ERelation \ (List\, A) \ (List\, A)$$

$$subseq'\, A \equiv cataListERel\, A \ (List\, A) \ (eRelIsNil\, A) \ (eRelIsCons\, A \cup = \ outrRel\, A \ (List\, A))$$

We can prove that the first, pointwise version, is actually equivalent to the catamorphism version. We will show the definition of a full proof of equality by double inclusion, one field of the top signature for each inclusion.

A brief explanation of the structure of this proof is in order. We recall that the definition of the relational list catamorphism E-relation consisted of an existential, inside of which there was a conjunction, and the second conjunct was in turn a relational composition. Each of these levels is represented, by the Curry-Howard isomorphism, by a proof object of a binary product type (a **sig**). The values of each nested proof object (the **structs** below) fill all the required components for the catamorphism to hold. The proof consists mainly of moving the appropiate pars of the proof object for the intuitive definition of the subsequence relation to the parts of the catamorphism proof object that need to be inhabited.

$$thEqSubseqs \ (A \ \in \ Setoid) \in subseqERel\, A = subseqERel'\, A$$

$$thEqSubseqs\, A \equiv$$

```
struct
  [ fst ≡ λ l m h →
      case l of
        [] →  case m of
                [] → tt
                x : xs → case h of
        x : xs →
          case m of
            [] → struct
                   [ fst ≡ []
                   [ snd ≡ struct
                             [ fst ≡ (thEqSubseqs A) .fst xs [] tt
                             [ snd ≡ inr tt
            x′ : xs′ →
              case h of
                inl hdEq →
                  struct
                    [ fst ≡ xs′
                    [ snd ≡ struct
                              [ fst ≡ (thEqSubseqs A) .fst xs xs′ hdEq.snd
                              [ snd ≡ inl ( struct
                                              [ fst ≡ hdEq.fst
                                              [ snd ≡ (List A) .ref xs′ )
                inr hdNEq →
                  struct
                    [ fst ≡ m
                    [ snd ≡ struct
                              [ fst ≡ (thEqSubseqs A) .fst xs m hdNEq
                              [ snd ≡ inr ((List A) .ref m)
  [ snd ≡ λ l m h →
      case l of
        [] →  case m of
                [] → tt
                x : xs → case h of
        x : xs →
          case m of
            [] → tt
            x′ : xs′ →
              case h.snd.snd of
                inl hdIn → inl ( struct
                                   [ fst ≡ hdIn.fst
                                   [ snd ≡ (thEqSubseqs A) .snd xs xs′
                                         ((subseqERel′ A) .resp xs xs h.fst xs′
                                           ((List A) .ref xs) hdIn.snd h.snd.fst) )
                inr hdNot → inr ( (thEqSubseqs A) .snd xs (x′ : xs′)
                                    ((subseqERel′ A) .resp xs xs h.fst (x′ : xs′)
                                      ((List A) .ref xs) hdNot h.snd.fst) )
```

An alternative definition, used in the book by Bird and de Moor, is based on a function that returns the list of all subsequences of its argument list. It can be seen as a functional implementation of the power relation of *subseqOf*.

$$subseqs \begin{pmatrix} A \in Setoid \\ l \in |\, List\, A\, | \end{pmatrix} \in |\, List\ (List\, A)\ |$$

$$subseqs\, A\, [\,] \equiv [\,]\ :\ [\,]$$

$$subseqs\, A\ (x\ :\ xs) \equiv append\ (map\, \lambda\, m\ \rightarrow\ x\ :\ m\ (subseqs\, A\, xs))\ (subseqs\, A\, xs)$$

To prove that this function really behaves as the implementation of such power relation, first we need to prove some lemmas. The first one is quite specific to this situation, and deals with the interaction of the member relation for lists with the action of consing the same element in front of each list that is part of a list of lists.

$$lemMemMapCons \begin{pmatrix} A \in Setoid \\ a \in |\, A\, | \\ ls \in |\, List\ (List\, A)\ | \\ a' \in |\, A\, | \\ m \in |\, List\, A\, | \end{pmatrix} \in$$
$$member\ (map\, \lambda\, l\ \rightarrow\ a\ :\ l\, ls)\ (a'\ :\ m)\ \leftrightarrow\ (==\ A\, a\, a'\ \&\ member\, ls\, m)$$

$$lemMemMapCons\, A\, a\, ls\, a'\, m \equiv$$

```
struct
  [ fst ≡ λ h →
        case ls of
          [] →  case h of
          l : ls′ →
              case h of
                                   struct
                inl inHd →        [ fst ≡ inHd.fst
                                  [ snd ≡ inl inHd.snd
                                  [ aux ∈== A a a′ & member ls′ m
                              let [   aux ≡ (lemMemMapCons A a ls′ a′ m) .fst inTl
                inr inTl →          struct
                              in  [ fst ≡ aux.fst
                                  [ snd ≡ inr aux.snd
  [ snd ≡ λ h →
        case ls of
          [] →  case h.snd of
          l : ls′ →
              case m of
                [] →
                    case l of
                                   ( struct           )
                      [] → inl     (   [ fst ≡ h.fst  )
                                   (   [ snd ≡ tt     )
                      x : xs → inr ((lemMemMapCons A a ls′ a′ []).snd
                        ( struct                                 )
                        (   [ fst ≡ h.fst                        )
                        (   [           case h.snd of            )
                        (   [ snd ≡                              )
                        (   [             inr inTl → inTl        )
                x : xs →
                    case h.snd of
                                    case l of
                                      [] →  case inHd of
                      inl inHd →                       ( struct           )
                                      x′ : xs′ → inl   (   [ fst ≡ h.fst  )
                                                       (   [ snd ≡ inHd   )
                      inr inTl → inr ((lemMemMapCons A a ls′ a′ (x : xs)) .snd
                        ( struct           )
                        (   [ fst ≡ h.fst  )
                        (   [ snd ≡ inTl   )
```

The second lemma is more general, and establishes the relationship between member and append for lists.

$$
\mathit{lemMemAppend} \begin{pmatrix} A \in \mathit{Setoid} \\ l \in |\,\mathit{List}\,A\,| \\ m \in |\,\mathit{List}\,A\,| \\ a \in |\,A\,| \end{pmatrix} \in
$$
$$
\mathit{member}\ (\mathit{append}\,l\,m)\ a\ \leftrightarrow\ (\mathit{member}\,l\,a\ \vee\ \mathit{member}\,m\,a)
$$

$$
\mathit{lemMemAppend}\,A\,l\,m\,a \equiv
$$

$$
\begin{array}{l}
\textbf{struct} \\
\left[\begin{array}{l}
\mathit{fst} \equiv \lambda\,h \rightarrow
\begin{array}{l}
\textbf{case}\ l\ \textbf{of} \\
\quad [\,] \rightarrow \\
\qquad \textbf{case}\ m\ \textbf{of} \\
\qquad\quad [\,] \rightarrow\ \textbf{case}\ h\ \textbf{of} \\
\qquad\quad x : xs \rightarrow \textbf{inr}\,h \\
\quad x : xs \rightarrow \\
\qquad \textbf{case}\ h\ \textbf{of} \\
\qquad\quad \textbf{inl}\,\mathit{inHd} \rightarrow \textbf{inl}\ (\textbf{inl}\,\mathit{inHd}) \\
\qquad\quad \textbf{inr}\,\mathit{inTl} \rightarrow
\begin{array}{l}
\textbf{case}\ (\mathit{lemMemAppend}\,A\,xs\,m\,a)\,.\mathit{fst}\ \mathit{inTl}\ \textbf{of} \\
\quad \textbf{inl}\,\mathit{hL} \rightarrow \textbf{inl}\ (\textbf{inr}\,\mathit{hL}) \\
\quad \textbf{inr}\,\mathit{hR} \rightarrow \textbf{inr}\,\mathit{hR}
\end{array}
\end{array}
\end{array}\right. \\
\left[\begin{array}{l}
\mathit{snd} \equiv \lambda\,h \rightarrow \\
\quad \textbf{case}\ l\ \textbf{of} \\
\qquad [\,] \rightarrow
\begin{array}{l}
\textbf{case}\ h\ \textbf{of} \\
\quad \textbf{inr}\,\mathit{hR} \rightarrow \mathit{hR}
\end{array} \\
\qquad x : xs \rightarrow \\
\qquad\quad \textbf{case}\ h\ \textbf{of} \\
\qquad\qquad \textbf{inl}\ (\textbf{inl}\,\mathit{inHd}) \rightarrow \textbf{inl}\,\mathit{inHd} \\
\qquad\qquad \textbf{inl}\ (\textbf{inr}\,\mathit{inTl}) \rightarrow \textbf{inr}\ ((\mathit{lemMemAppend}\,A\,xs\,m\,a)\,.\mathit{snd}\ (\textbf{inl}\,\mathit{inTl})) \\
\qquad\quad \textbf{inr}\,\mathit{hR} \rightarrow \textbf{inr}\ ((\mathit{lemMemAppend}\,A\,xs\,m\,a)\,.\mathit{snd}\ (\textbf{inr}\,\mathit{hR}))
\end{array}\right.
\end{array}
$$

We can now present the theorem regarding the connection between the function that returns the list of all subsequences and the two equivalent subsequence relations presented before. The property satisfied is that the subsequence relation holds between two lists iff the second list is a member of the result of applying the function to the first list.

$$
\mathit{thSubseqs2} \begin{pmatrix} A \in \mathit{Setoid} \\ l \in |\,\mathit{List}\,A\,| \\ m \in |\,\mathit{List}\,A\,| \end{pmatrix} \in \mathit{member}\ (\mathit{subseqs}\,A\,l)\ m\ \leftrightarrow\ \mathit{subseq}\,A\,l\,m
$$

$$
\mathit{thSubseqs2}\,A\,l\,m \equiv
$$

```
struct
  [ fst ≡ λ h →
      case l of
        [] →  case m of
                [] → tt
                x : xs →  case h of
        x : xs →
            case m of
              [] → tt
              x′ : xs′ →
                  case ( lemMemAppend
                         (List A)
                         (map λ l → x : l (subseqs A xs))
                         (subseqs A xs)
                         (x′ : xs′)
                       ).fst h of
                    inl inFst →
                        let [ aux ∈ == A x x′ & member (subseqs A xs) xs′
                              aux ≡ (lemMemMapCons A x
                                       (subseqs A xs) x′ xs′).fst inFst
                            ]
                          inl
                        in ( struct
                               [ fst ≡ aux.fst
                                 snd ≡ (thSubseqs2 A xs xs′).fst aux.snd ] )
                    inr inSnd → inr ((thSubseqs2 A xs (x′ : xs′)).fst inSnd)
  ]
  [ snd ≡ λ h →
      case l of
        [] →  case m of
                [] → inl tt
                x : xs →  case h of
        x : xs →
          case m of
            [] → (thSubseqs2 A (x : xs) []).snd tt
            x′ : xs′ →
              case h of
                inl hdEq → ( lemMemAppend
                             (List A)
                             (map λ l → x : l (subseqs A xs))
                             (subseqs A xs)
                             (x′ : xs′)
                           ).snd
                  ( inl
                    ( (lemMemMapCons A x (subseqs A xs) x′ xs′).snd
                      ( struct
                          [ fst ≡ hdEq.fst
                            snd ≡ (thSubseqs2 A xs xs′).snd hdEq.snd ] ) ) )
                inr hdNEq → ( lemMemAppend
                              (List A)
                              (map λ l → x : l (subseqs A xs))
                              (subseqs A xs)
                              (x′ : xs′)
                            ).snd
                    (inr ((thSubseqs2 A xs (x′ : xs′)).snd hdNEq))
  ]
```

Chapter 6

Relational Databases in Type Theory

6.1 Introduction

In the field of databases, one speaks about the *logical model* of data as opposed to the *physical model* of data: in the logical model we are only concerned with what data are stored in the database and with the relationships between such data. How it is stored, and how relationships are represented is left to the implementation of the database, that is, to the physical model. One of these logical data models is the *relational model*, which is based on the usual notion of relation in the mathematical sense.

Type theory is a good framework for relating the logical and physical models of data. On the logical side we have relations, which can be formalized in type theory as the previous chapters have explored. We can define all the fundamental concepts of the relational model in type theory by trying to formalize and pin down what the naive set theory definitions of those concepts mean constructively. In particular, we shall show how to deal with attributes and relation schemas, a powerful part of the relational model that is usually dealt with in a very general (and sometimes even vague) way. When we have done that, we can attempt to define all the basic operations of the model, for instance selection, projection and join. Then, we would like to show as part of the formalization the laws these operations satisfy.

On the physical side, we can formalize in type theory the data structures needed for implementing the relational model. These structures can range from the most simple ones (list-based, intuitively corresponding to the idea of sequential files) to those actually used in the implementation of state-of-the-art database management systems (for instance, dynamic hash tables or B trees). With a formal definition of the data structures to be used, we can then define how the relational model's operations are implemented in terms of these data structures and their associated algorithms. Following the well established idea of using type theory to prove programs correct, an ambitious goal at that point would be to prove that the implementations of the relational operations by the concrete data structures satisfy the laws of those relational operations at the logical data level.

In this chapter we attempt a simple beginning of the programme of establishing what is the formal basis of relational database theory in constructive type theory. All our work is carried out in the proof editor Alfa for Martin-Löf's monomorphic type theory. Our goal is to obtain a formalisation that captures as much as possible the intuitive presentation in textbooks of relational database theory. Hence we present a series of formalizations of increasing degree of complexity, to explore the points at which turning the naive set theoretic explanation of the relational model into formal type theory implies fundamental decisions on the representations to be used. As will be seen, there is a certain trade-off between how much of the full relational model we want to capture, and how simple and elegant the formalization becomes.

For the purpose of the present work, the previously mentioned desired goal of proving implementations of relational databases correct will only be done in some simple list-based structures. In the future, we hope to achieve similar results for more realistic structures.

6.2 Set-theoretic databases

We now present a survey of the basic standard concepts of relational database theory. This will not only introduce the relational model of data, but also fix the precise meaning we will give to its concepts for the rest of this chapter, and so serve as a reference for comparing the several formalizations of these concepts in type theory. We follow standard notation in this section, which is the one used in many textbooks and other literature on relational databases. In particular, see the book by Maier [53], since it is the work we have followed most closely on such matters, and where proofs of the properties mentioned in this section can also be found.

6.2.1 Basic concepts

Domains

Definition A *data domain* is a (usually finite) set of values.

Discussion Domains can be any data type, equipped with operations on the datatype that are suitable for its use. In general, a boolean-valued equality test is part of those operations, for all domains in the database.

Attributes

Definition An *attribute name* is an identifying label for an atomic piece of data. The set of all attributes is denoted by $\mathcal{A}$.

Discussion Attributes are usually descriptive names, for instance *Telephone-number*. One can think of $\mathcal{A}$ as the set of all character strings. Arbitrary attributes are usually denoted by capital letters, such as A, B, X. Attribute names are usually "global", in the sense that they can be used in any part of a database.

Domain of attributes

Definition The *domain of an attribute A*, denoted $\Delta(A)$, is some data domain.

Discussion Δ is a function mapping attribute names to data domains. This function is defined globally, that is, the domain associated with an attribute is given once and for all when Δ is given.

Schemes, ordered

Definition A *relation scheme* is a finite list of different attribute names. For a scheme $R = (A_1, \ldots, A_n)$, we call the cartesian product $\Delta(R) = \Delta(A_1) \times \ldots \times \Delta(A_n)$ its *typing*.

Discussion A scheme is like the heading of a table used to display a relation, where each row of the table shows a different tuple of the relation. Relations are defined with respect to schemes, instead of directly in terms of the corresponding domains. One reason for this is a more intuitive design (since attributes can have descriptive names that better reflect the real-world semantics; for instance, *"Telephone-number"* instead of just a subset of the naturals). A second reason is that attribute names are used to keep all relations of a database connected: the appearance of the same attribute name in two different schemes will refer to the same real-world property of interest.

Relations, ordered

Definition A *relation r over* scheme R (denoted for short $r(R)$) is a subset of the cartesian product $\Delta(R)$. An element of r is called a *tuple*.

Discussion Relations in this sense coincide with the usual mathematical idea of relation. While in the previous chapters we have dealt with binary relations only, now we allow relations of any arity since they are a more natural way of representing data. A more significant difference than the extended arities is that a relation is now defined via a scheme, instead of directly on a cartesian product of domains. One can also say that a relation is a subset of an indexed cartesian product $\Pi_{i\in\{1,\ldots,n\}}\Delta(A_i)$. Usually one thinks of the scheme as the (name of the) real type of a relation. When defining database operations, the typing restrictions are expressed in terms of schemes.

Ordered or not?

Looking at relations in the above way makes use of an ordering of the components of a tuple that is not quite natural, since there is no a priori ordering on the elements of a finite set. Once one considers the presence of attributes, the usual way to refer to one such component for a tuple t would be by the attribute name of the desired component. That is, a tuple can be thought of as a record, the order or the field labels being irrelevant. We would use for instance the notation $t[A]$ for the value of the field A of t, where A is a single attribute name.

This becomes more of a real problem when it comes to defining database operations. Many useful operations depend on the attributes present in a scheme, and not on the order those attributes are listed in the scheme. One could choose to either restrict the operations to those in which the schemes are in some particular order, or take into account all permutations of the scheme. For instance, thinking about the union of two relations (seen as sets of tuples), we could only allow it if the relations are over the same scheme (which would imply that the attributes are in the same order); or we could allow it for any relations whose schemes are permutations of each other. The first way is very common in actual database management systems, but many concepts of relational database theory are actually better expressed in an unordered-scheme way. Instead of dealing with permutations, we can change the definitions to remove the arbitrary orderings on the schemes.

Schemes, unordered

Definition A *scheme* is a finite set of attribute names. For a scheme $R = \{A_1, \ldots, A_n\}$, we call its *typing* the function Δ that assigns to $A \in R$ the set $\Delta(A)$.

Tuples, unordered

Definition A *tuple over* scheme R is a function that for each $A \in R$ gives a value $v \in \Delta(A)$. We denote the value of t on attribute $A \in R$ by $t[A]$.

Relations, unordered

Definition A *relation* r *over* scheme R (denoted for short $r(R)$) is a set of tuples over R.

Tuple projection, unordered

Definition For a tuple t over scheme R, and $S \subseteq R$, the *projection* of t over S (denoted $t[S]$) is the restriction of the function for t to S.

Discussion This idea is very useful for defining database operations, and also is fundamental for easily defining notions of data dependency theory. Defining an analogous idea for the ordered case is not so natural, and is the reason why the aforementioned theory assumes schemes to be sets and not lists.

Databases

Definition For a set of attributes $\mathcal{A}$ and a function Δ associating attributes to domains, a *database* is a set of relations over schemes mentioning only attributes in $\mathcal{A}$.

6.2.2 Operations

Relational algebra

A *query* is an expression consisting of operations, relations and attributes forming part of a database, and returning as its result another relation. Its purpose is retrieving some information of interest for the user of the database. There are formal languages in which one can form such expressions, and the one we will deal with is called *relational algebra*. The operations it provides can be divided into set operations and special relational operations.

Set operations

Definition For relations $r_1(R)$ and $r_2(R)$, their *union* $r_1 \cup r_2$, *intersection* $r_1 \cap r_2$ and *difference* $r_1 \setminus r_2$ are respectively just their union, intersection and difference when considering relations as sets of tuples.

Discussion It should be evident that the preceding definition does not depend on whether schemes are lists or sets of attributes. The usual set-theoretical properties hold for these operations, and $\emptyset$ will denote the empty relation.

Projection, unordered

Definition For any relation $r(R)$ and $S \subseteq R$, the *projection* of r over S (denoted $\pi_S(r)$) is the relation over scheme S given by $\pi_S(r) = \{t[S] \mid t \in R\}$.

Discussion Projection operations are typically used to cut off parts of a relation that are not of interest for finding the result of a query.

Selection, unordered

Definition For a relation $r(R)$ and a predicate P over tuples over R, the result of the *selection* operation (denoted $\sigma_P(r)$) is the relation over scheme R given by $\sigma_P(r) = \{t \mid P(t)\}$.

Discussion Selection operations are used for searching for information in a relation. The predicate P expresses the condition of interest in a query, and can use constants, attribute names in R, tuple projections, comparisons and logical connectives.

Natural join, unordered

Definition For any relations $r(R)$ and $s(S)$, their *natural join* $r \bowtie s$ is the relation over scheme $R \cup S$ given by $r \bowtie s = \{t \mid \exists x \in r, y \in s.\ (t[R] = x \ \wedge t[S] = y)\}$.

Discussion Joins are related to the usual notion of relation composition. One difference is that the relations involved are not in general binary, but of any possible arity. Another difference is that in a composition the resulting tuple dosen't contain the "bridging" elements as they do in the result of a join. The composition has result scheme $(R \cup S) - (R \cap S)$, while the join has result scheme $R \cup S$. Two tuples in the argument relations will produce a tuple of the join result if they agree on their values for all attributes in $R \cap S$. The usefulness of joins from the point of view of data management is that they allow us to put together information stored in different relations. Given this intended use, joins are quite naturally used to express integrity constraints of several kinds, too.

Theorem

The relational database operations satisfy the following laws, called *equivalence rules*:

- $\sigma_{\Theta_1 \wedge \Theta_2}(E) = \sigma_{\Theta_1}(\sigma_{\Theta_2}(E))$

- $\sigma_{\Theta_1}(\sigma_{\Theta_2}(E)) = \sigma_{\Theta_2}(\sigma_{\Theta_1}(E))$
- $\pi_{L_1}(\pi_{L_2}(E)) = \pi_{L_1}(E)$, for $L_1 \subseteq L_2$
- $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$
- $E \bowtie E = E$
- $E_1 \bowtie E_2 = E_2 \bowtie E_1$
- $\sigma_P(E_1\,\Omega\,E_2) = \sigma_P(E_1)\,\Omega\,\sigma_P(E_2)$, for $\Omega = \cup, \cap, -$
- $\sigma_P(E_1\,\Omega\,E_2) = \sigma_P(E_1)\,\Omega\,E_2$ for $\Omega = \cap, -$
- $\pi_X(\sigma_P(E)) = \sigma_P(\pi_X(E))$ if P only mentions attributes in X
- $\pi_L(E_1 \cup E_2) = \pi_L(E_1) \cup \pi_L(E_2)$
- $\pi_L(E_1 \bowtie E_2) \subseteq E_1$ if L is the scheme of E_1
- $E \subseteq \pi_L(E) \bowtie \pi_M(E)$ if $L \cup M$ is the scheme of E
- $(E_1 \cup E_2) \bowtie E_3 = (E_1 \bowtie E_3) \cup (E_2 \bowtie E_3)$
- $\sigma_P(E_1 \bowtie E_2) = \sigma_P(E_1) \bowtie E_2$ if P only mentions attributes in E_1
- $\sigma_{\Theta_1 \wedge \Theta_2}(E_1 \bowtie E_2) = \sigma_{\Theta_1}(E_1) \bowtie \sigma_{\Theta_2}(E_2)$ if Θ_1 mentions only attributes in E_1 and Θ_2 mentions only attributes in E_2
- $\emptyset$ is a zero of all joins, projections and selections

Other database operations

It is common to have available some additional relational operations. Some of them are easily defined in terms of the ones we presented (for instance, cartesian product can be obtained by a join and renaming of any common attributes), others have more complex definitions (for instance, relation division), and there are even some that are a proper extension of the basic set of operations (usually called *aggregate operators*, which perform such things as calculating averages, maxima, or cardinalities). In this work we will not address the last kind of operations, and the other operations will be assumed to always be definable in terms of these more basic ones.

6.2.3 Data dependencies

Integrity constraints

Besides being type correct (in the sense that tuples stored in relations should have components of the domains associated to their scheme's attributes), a database needs to capture additional real-world semantics. Consistency properties of the information stored in the database can be expressed in several ways. One common and useful way is by using data dependencies. These have a rich formal theory and are quite often used in practice.

Functional dependency

Definition A functional dependency on scheme R is a pair (α, β), with $\alpha, \beta \subseteq R$. The dependency will be denoted as $\alpha \rightarrow \beta$.

Holds A functional dependency $\alpha \rightarrow \beta$ over scheme R *holds* on relation $r(R)$ if for any tuples $t_1, t_2 \in r$, if $t_1[\alpha] = t_2[\alpha]$ then $t_1[\beta] = t_2[\beta]$.

Discussion Many approaches to properly decomposing the total information on a database into a set of relations are based on functional dependencies. These *normal forms* typically require that all dependencies comply with a particular shape or contents of their attributes.

Keys

Superkey A *superkey* for scheme R is a $K \subseteq R$ such that $K \rightarrow R$ holds on R.

A superkey uniquely identifies tuples in any possible relation over a scheme. The definition simply says that no two different tuples can have the same values on all attributes that form the superkey.

Candidate key A *candidate key* K for scheme R is a minimal superkey for R. That is, there is no $K' \subset K$ such that K' is also a superkey for R.

It is standard practice to specify with each relation scheme its candidate keys. For implementation purposes, database managers typically require one to name a particular candidate key as the scheme's *primary key*. This primary key will be used heavily to create indices and other data structures that allow efficient access to the relation's contents.

6.3 About the formalization of relational database theory

We shall now discuss how to formalize the fundamental concepts of relational database theory in Martin-Löf's type theory. The first thing to be noted about this is that it is not straightforward. The basic reason is that the notion of "set" in classical set theory is very flexible, but not very computational. This flexibility is used to a great extent to set up very elegant and intuitive-looking definitions for database theory, as the preceding section hopefully illustrated. When one actually becomes concerned with implementing "sets" in a programming language, a variety of methods are available. For example, the various kinds of "collections" provided by data structure libraries.

In type theory we find several different notions which may be used for representing the classical sets. We have the basic type-theoretic notion of "set", meaning "data type" in a similar sense to the one used in functional programming languages. We also have the notion of "setoid", meaning a datatype with a specific notion of equality defined on it. We have the notion of "datoid", which is a datatype with a substitutive and decidable equality. We have "finite sets", which can be defined in several different ways. And finally, we may want to take an abstract approach and just give axioms for finite sets, which later may be implemented by efficient data structures (for instance, hash tables, B-trees, etc.) also expressed inside type theory.

Our intention is to benefit from the fact that Martin-Löf's type theory is a language both for mathematics (a "constructive set theory") and a programming language. The mathematics is developed much as in a standard database textbook, although the set-theoretic notions need to be modified to become appropriate type-theoretic notions. The programming notions (algorithms, data structures) are also developed inside type theory, in much the same way as they would be developed in a programming language. Of course, since type theory is a functional programming language, the data structures would be developed as in e.g. Okasaki's [56] approach, rather than in a more traditional account carried out in an imperative language.

Once we have unified the set-theory and the programming parts, we are in a much better position to provide formal correctness proofs of the database algorithms. These proofs would be developed in a classical approach, where the database algorithms are interpreted in set theory using the (probably complex) denotational semantics of the programming language. It does not seem feasible to carry out formal correctness proofs (in a proof tool for set theory) in such an approach, whereas we hope that it would be feasible to carry out correctness proofs in our setting. This remains to be shown, however.

6.4 General indexed relations

We begin by discussing "general indexed relations", which is the most general notion and can be viewed as the type-theoretic analogue of the set-theoretic relations used in classical accounts of database theory.

General indexed relations are propositional functions on a set-indexed family of setoids. The indexing set corresponds to the database "scheme" and the setoids are the types of values of entries in the tuples that form a relation from the database. We need setoids rather than sets because the database operations require us to compare entries with respect to the equality on their corresponding types.

6.4.1 Schemes and attributes

A scheme will be some *Set*:

$Scheme \in Type$

$Scheme \equiv Set$

No assumptions about its structure are needed, in particular nothing is said about a scheme being ordered or not. Also, note that a scheme is not a setoid in this approach. An attribute A of a scheme I is just an element $A \in I$.

6.4.2 Domains

The domains associated with attributes will be setoids. This is required, instead of being simply sets, because the definitions of several fundamental operations (like projections and joins) involve equality comparisons between tuples and subtuples, and these in turn depend on equality comparisons between values of corresponding positions in the tuples. Hence, each attribute domain must provide such an equality. Working with setoids is much like working with types of the Eq-class in Haskell.

The association of attributes and their domains is not done globally, as one would expect (a global attribute typing function is typically the case in relational database theory), but at the level of the schemes. Hence, if I is a scheme, for each element of I there will be a setoid A_i. We give a name to the type of all such associations:

$Typing\ (I \in Set) \in Type$

$Typing\ I \equiv I \rightarrow Setoid$

6.4.3 Tuples

A tuple over a scheme takes as argument not just the scheme, but also the whole association function just described. The tuple itself is a dependently typed function from elements of the scheme to elements of the associated carrier. So in this view, tuples are elements of the product of the indexed family of types $|A\,i|$.

$Tuple\ (I \in Scheme, A \in Typing\, I) \in Set$

$Tuple\, I\, A \equiv (i \in I) \to |\, A\, i\, |$

Tuple equality is the point-wise equality at the level of the associated setoids. Its definition is:

$eqTuple \in Rel\ (Tuple\, I\, A) \qquad (I \in Scheme, A \in Typing\, I)$

$eqTuple \equiv \lambda\, x\, y \to \forall\, I\, \lambda\, i \to (x\, i) ==_{(A\, i)} (y\, i)$

This means two tuples are equal if and only if they are extensionally equal as functions. Note also the use of infix notation for the equality: we will resort to this syntactic presentation feature of Alfa when it makes things clearer.

6.4.4 Relations

Database relations are propositional functions over tuples:

$Relation\ (I \in Scheme, A \in Typing\, I) \in Type$

$Relation\, I\, A \equiv Tuple\, I\, A \to Prop$

This gives us immediately the set operations over relations, in the usual type-theoretic way (union by disjunction, intersection by conjunction, difference by intersection and negation).

$unionRel\ (R, S \in Relation\, I\, A) \in Relation\, I\, A \qquad (I \in Scheme, A \in Typing\, I)$

$unionRel\, R\, S \equiv \lambda\, x \to R\, x \vee S\, x$

$intersRel\ (R, S \in Relation\, I\, A) \in Relation\, I\, A \qquad (I \in Scheme, A \in Typing\, I)$

$intersRel\, R\, S \equiv \lambda\, x \to R\, x\, \&\, S\, x$

Another operation that is easy to define is selection, since it just amounts to taking the conjunction of the selection predicate and the propositional function that represents the relation.

$selectRel\ (P \in Pred\ (Tuple\, I\, A), R \in Relation\, I\, A) \in Relation\, I\, A$
$(I \in Scheme, A \in Typing\, I)$

$selectRel\, P\, R \equiv \lambda\, x \to R\, x\, \&\, P\, x$

6.4.5 Projections

Since we just represent schemes as elements of *Set*, there is no immediate way of manipulating schemes and tuples for defining the operations that depend essentially on the structure of the scheme. One way of putting structure on a scheme is by using sum types: a scheme can then be of the form $I+J$, for $I, J \in Set$. To produce the typing corresponding to the sum of the two original typings of the schemes, we use this function:

$$add2Typings\ (A \in Typing\ I, B \in Typing\ J) \in Typing\ (Plus\ I\ J) \qquad (I, J \in Scheme)$$

$$add2Typings\ A\ B \equiv \lambda\, k \to \begin{array}{l} \textbf{case}\ k\ \textbf{of} \\ \quad \textbf{inl}\ i \to A\ i \\ \quad \textbf{inr}\ j \to B\ j \end{array}$$

This way of structuring schemes by sum types gives us two projection functions:

$$projRel1\ (R \in Relation\ (Plus\ I\ J)\ (add2Typings\ A\ B)) \in Relation\ I\ A$$
$$(I, J \in Scheme, A \in Typing\ I, B \in Typing\ J)$$

$$projRel1\ R \equiv \lambda\, x \to \exists y \in$$
$$Tuple\ (Plus\ I\ J)\ (add2Typings\ A\ B)\ .\ R\ y\ \&\ eqTuple\ (projTuple1\ A\ B\ y)\ x$$

$$projRel2\ (R \in Relation\ (Plus\ I\ J)\ (add2Typings\ A\ B)) \in Relation\ J\ B$$
$$(I, J \in Scheme, A \in Typing\ I, B \in Typing\ J)$$

$$projRel2\ R \equiv \lambda\, x \to \exists y \in$$
$$Tuple\ (Plus\ I\ J)\ (add2Typings\ A\ B)\ .\ R\ y\ \&\ eqTuple\ (projTuple2\ A\ B\ y)\ x$$

6.4.6 Joins

Using the same idea of introducing structure on the schemes by sum types, we can define a similar, restricted, version of join. The schemes will now be structured as a sum type with three cases:

$$Triple\ (A, B, C \in Set) \in Set$$

$$Triple\ A\ B\ C \equiv \begin{array}{l} \textbf{data} \\ \quad \textbf{fst}\ (a \in A) \\ \quad \textbf{snd}\ (b \in B) \\ \quad \textbf{trd}\ (c \in C) \end{array}$$

For schemes, the order in which the sums are applied to create the new type is immaterial, but certainly has to be fixed once and for all. The corresponding function to construct the typing for a triple sum from the typings of the three original schemes is the following:

$add3Typings\ (A \in Typing\ I, B \in Typing\ J, C \in Typing\ K) \in Typing\ (Triple\ I\ J\ K)$
$(I, J, K \in Scheme)$

$$add3Typings\ A\ B\ C \equiv \lambda\, l \;\rightarrow\; \begin{array}{l}\textbf{case}\ l\ \textbf{of} \\ \quad \textbf{fst}\ a \rightarrow A\ a \\ \quad \textbf{snd}\ b \rightarrow B\ b \\ \quad \textbf{trd}\ c \rightarrow C\ c\end{array}$$

The definition of the join of two tuples whose schemes are sum types with one of the parts in common is as follows:

$joinRel$
$(R \in Relation\ (Plus\ I\ J)\ (add2Typings\ A\ B), S \in Relation\ (Plus\ J\ K)\ (add2Typings\ B\ C))$
$\in Relation\ (Triple\ I\ J\ K)\ (add3Typings\ A\ B\ C)$
$(I, J, K \in Scheme, A \in Typing\ I, B \in Typing\ J, C \in Typing\ K)$

$joinRel\ R\ S \equiv \lambda\, x \rightarrow$
$\quad (\exists\, y \in Tuple\ (Plus\ I\ J)\ (add2Typings\ A\ B)\ .\ R\ y\ \&\ eqTuple\ (projTriTup1\ A\ B\ C\ x)\ y)\ \&$
$\quad (\exists\, z \in Tuple\ (Plus\ J\ K)\ (add2Typings\ B\ C)\ .\ S\ z\ \&\ eqTuple\ (projTriTup2\ A\ B\ C\ x)\ z)$

In this definition, two auxiliary functions are used. They take a tuple formed by a triple sum, and return the tuples corresponding respectively to the first two and last two parts of the triple sum:

$projTriTup1$
$(A \in Typing\ I, B \in Typing\ J, C \in Typing\ K, x \in Tuple\ (Triple\ I\ J\ K)\ (add3Typings\ A\ B\ C))$
$\in Tuple\ (Plus\ I\ J)\ (add2Typings\ A\ B) \qquad (I, J, K \in Scheme)$

$$projTriTup1\ A\ B\ C\ x \equiv \lambda\, l \;\rightarrow\; \begin{array}{l}\textbf{case}\ l\ \textbf{of} \\ \quad \textbf{inl}\ i \rightarrow x\ (\textbf{fst}\ i) \\ \quad \textbf{inr}\ j \rightarrow x\ (\textbf{snd}\ j)\end{array}$$

$projTriTup2$
$(A \in Typing\ I, B \in Typing\ J, C \in Typing\ K, x \in Tuple\ (Triple\ I\ J\ K)\ (add3Typings\ A\ B\ C))$
$\in Tuple\ (Plus\ J\ K)\ (add2Typings\ B\ C) \qquad (I, J, K \in Scheme)$

$$projTriTup2\ A\ B\ C\ x \equiv \lambda\, l \;\rightarrow\; \begin{array}{l}\textbf{case}\ l\ \textbf{of} \\ \quad \textbf{inl}\ j \rightarrow x\ (\textbf{snd}\ j) \\ \quad \textbf{inr}\ k \rightarrow x\ (\textbf{trd}\ k)\end{array}$$

6.4.7 Proving properties

Since the set-theoretic operations on relations are defined directly in terms of the logical constants, proofs involving only them can be established by resorting to the properties of the corresponding constants. Properties of selection (alone or interacting with the set-theoretic operations) can also be established in that way, since selection is just defined in terms of a conjunction in this approach.

Proofs of properties involving projections and joins are not so straightforward. One needs to deal with all the assembling and disassembling of schemes and tuples through the several sum types that may be involved in the particular property.

6.4.8 Dependencies

Our approach to sum types can still work for defining the notion of superkey. We simply need to take a scheme that is a sum, its two parts being the key attributes and the non-key attributes. The definition proceeds by universal quantification over the sum type tuples:

$$\begin{array}{l} \mathit{superkey}\ (R \in \mathit{Relation}\ (\mathit{Plus}\, I\, J)\ (\mathit{add2Typings}\, A\, B)) \in \mathit{Prop} \\ (I, J \in \mathit{Scheme}, A \in \mathit{Typing}\, I, B \in \mathit{Typing}\, J) \end{array}$$

$$\begin{array}{l} \mathit{superkey}\, R \equiv \forall\ (\mathit{Tuple}\ (\mathit{Plus}\, I\, J)\ (\mathit{add2Typings}\, A\, B))\ \lambda\, t1\ \rightarrow \\ \forall\ (\mathit{Tuple}\ (\mathit{Plus}\, I\, J)\ (\mathit{add2Typings}\, A\, B))\ \lambda\, t2\ \rightarrow \\ \mathit{eqTuple}\ (\mathit{projTuple1}\, A\, B\, t1)\ (\mathit{projTuple1}\, A\, B\, t2)\ \supset \\ \mathit{eqTuple}\ (\mathit{projTuple2}\, A\, B\, t1)\ (\mathit{projTuple2}\, A\, B\, t2) \end{array}$$

A similar approach could be used for defining a functional dependency $I \rightarrow J$ for $I \cap J = \emptyset$. If there are attributes in the scheme that are neither in I nor in J, we can still do something similar but with a sum of the form $\mathit{Triple}\ I\ J\ K$ (where K would represent all such attributes that are not part of the dependency).

However, this is not enough if we wish to deal with some of the richer concepts of data dependency. For instance, to define the notion of candidate key we would need the ability to talk about subschemes in general, so we could then talk about a minimal such subscheme. Also, to deal with dependencies in which the attributes involved don't partition the scheme completely, or have common attributes on their left and right hand sides, this approach is also insufficient by itself. We will now address these problems by exploring a modification to the current approach.

6.4.9 Representing subschemes

We will now address a way of using general indexed relations, while being able to talk about subschemes (and hence, general functional dependencies too). The representation of schemes and their typings will be the same as we have discussed so far in this section, for a start. We can talk about subsets in type theory by means of propositional functions. Thus, if $I \in \mathit{Set}$ is a scheme, a subscheme of it can be an element of the type $I \rightarrow \mathit{Prop}$. Thus, $J \in I \rightarrow \mathit{Prop}$ represents the subscheme of elements $i \in I$ such that $J\, i$ is true. A pair of two such elements can then be a representation of a functional dependency on scheme I.

Since this would make subschemes elements of $I \to Prop$ instead of Set, a few extra things are needed to keep close to our original definitions. First, we can represent the "extensions" of subschemes in this way:

$$extSubsch\ (I \in Set)(A \in I \to Prop) \equiv \Sigma a \in I.\ A\,a$$

Here we use a dependent pair type, as denoted by Σ. Elements of $extSubsch\ I\ A$ are pairs (a, p), where a is an attribute in the scheme, and p is a proof that a is in the subscheme. For brevity, we will write such an extension in the form $\Sigma\, I\, A$, in the understanding that $A \in I \to Prop$. (It is important to remark that if $A\,a$ may have more than one element, this will not be the intended notion of extension of a subscheme.)

We can represent typings for such extensions in this way:

$$Typing\ I\,(\Sigma IA) \equiv \Sigma\, I\, A \to Setoid$$

and write for brevity $T\,I\,A$ for it, for such a typing associated to a $\Sigma\, I\, A$. (Going back to the issue of potentially not having the intended notion, it could become a problem if we apply the typing function to two pairs (i, p), (i, p') with different proof objects on the second component. The problem would be that the setoids associated to i may be distinct if we don't require some additional properties as part of the definition. In the next subsection we will try to address this issue.)

Tuples will now be:

$$Tuple\ I\,(\Sigma\, I\, A)\,(T\, I\, A) \equiv (x \in \Sigma\, I\, A) \to |(T\, I\,(Ax))|$$

A relation would be represented like this:

$$Rel\ I(\Sigma\, I\, A)(T\, I\, A) \equiv Tuple\ I\,(\Sigma\, I\, A)\,(T\, I\, A) \to Prop$$

Definitions of the set operations over relations would proceed just as before. Selection is also directly definable in terms of a conjunction. To show the typical type of one of these operations that are homogeneous on the schemes involved:

$$interRel\ I\,(\Sigma A)\,(TA)(R, S \in Rel\ I\,(\Sigma A)\,(TA)) \in Rel\ I\,(\Sigma A)\,(TA)$$

To define the relational operations, we make use of the extensions. Given subschemes $A, B : I \to Prop$, their union $A \cup B : I \to Prop$ is just the propositional conjunction of them. So we can get, without any changes to this approach, the extension $\Sigma\, I\,(A \cup B)$ and the associated typing $T\, I\,(A \cup B)$. With this we can now define the join of $r \in Rel\ I\,(\Sigma\, I\, A)\,(T\, I\, A)$ and $s \in Rel\ I\,(\Sigma\, I\, B)\,(T\, I\, B)$ would have type $Rel\ I\,(\Sigma\, I\,(A \cup B))\,(T\, I\,(A \cup B))$.

Regarding projections, we can define an arbitrary projection operation. For this we need to first define a test for B being a subscheme of a scheme A, which is straightforward by using propositional implication. Using this same test, we can talk meaningfully about arbitrary functional dependencies over a relation scheme, and so define notions of superkeys and even candidate keys. This shows we can really define all the required operations in this modified representation, while now having the ability to deal with functional dependencies in all their generality.

6.4.10 Representing subschemas, with global typing

As mentioned above, the interaction of a local typing for attributes of a subscheme, with the potential multiple representations of the elements of the extension of this subscheme, can be problematic. (This happens if we don't have proof-irrelevance, i.e. when we have that $(i, p), (i, p') \in \Sigma I A$ implies $p = p'$, where this last equality is on the type of identity proofs.) A possible way out of this is if we have a universe U of attribute names, and we have a global typing function T of type $U \to \mathit{Setoid}$. In that case $T\, I\, A$ behaves nicely again, since it would be defined as

$$T\, I\, A \equiv \lambda x \in \Sigma I A. T\, I\, (x.\mathit{fst})$$

and so the potential distinct proof objects on the second component of x are ignored. Note that in this case both schemes and subschemes are represented by propositional functions.

The headings of the definitions change a bit now, since the typing is global and hence TA is not an argument anymore. For instance,

$$\mathit{Tuple}\, I\, (\Sigma I A) \equiv (x \in \Sigma I A) \to |T\, I\, (x.\mathit{fst})|$$

A relation would be represented as:

$$\mathit{Rel}\, I(\Sigma I A) \equiv \mathit{Tuple}\, I\, (\Sigma I A) \to \mathit{Prop}$$

For the operations, the typing arguments again disappear, which makes the headings more readable as a side benefit.

6.5 Finite-indexed, decidable relations

The general indexed relations are appropriate type-theoretic analogues of the general (possibly infinite) set-theoretic relations used in the classical foundation of relational database theory. But although our type-theoretic general indexed relations are constructive, they nevertheless do not capture the computational nature of relational databases. Database operations are computable, because the attribute sets and the relations are finite, and equality of database entries is decidable (typically equality defined on strings or integers). To capture this situation we consider a formalization of finite-indexed, decidable relations.

6.5.1 Attributes

Attributes will now be elements of a finite set. We will use the usual finite sets that appear in type theory, the N_n sets, $n \geq 0$. These are defined informally by $N_n = \{0, 1, \ldots, n-1\}$. Following the Alfa library conventions, we will write them down as $\mathit{Fin}\, n$, for $n \in \mathit{Nat}$ and Nat the inductively defined types of natural numbers.

A consequence of this choice is that attributes are no longer descriptive, which was a possibility in the previous approach. Now they are just positional indices, as we will see. In particular, we still have the same situation as in the previous approach: there is no global set of attributes, and no global association of attributes to data domains.

6.5.2 Schemes

A scheme will be a particular *Fin n*, where *n* is the arity of the tuples that form part of relations over this scheme. We only need to know *n* to know which scheme it is:

$Scheme \in Set$

$Scheme \equiv Nat$

The attributes in a scheme will be, as mentioned before, positional indices into it.

6.5.3 Domains

The domains associated with attributes are now datoids, that is, sets with a decidable substitutive equality. The reason for this is that we want all the database operations to be decidable. As can be seen in the preceding section on general relations, some of those operations involve comparing tuples. Since tuple comparison must then be decidable and it is defined in terms of tuple component comparison, this must also be decidable. Hence, all the data domains must have decidable equalities, that is, they are Boolean valued functions.

As in the preceding approach, the association is not done globally but at the level of the schemes. The type of all such associations is thus represented:

$Typing\ (n \in Scheme) \in Type$

$Typing\ n \equiv Fin\ n \rightarrow Datoid$

6.5.4 Tuples

As in the preceding approach, a tuple over a scheme also takes a typing as an argument, and is a function taking attributes of the scheme to elements of the associated carrier:

$Tuple\ (n \in Scheme, A \in Typing\ n) \in Set$

$Tuple\ n\ A \equiv (i \in Fin\ n) \rightarrow |\, A\ i\,|$

Tuple equality is again the element-wise equality at the level of the associated datoids. But its type is now boolean:

$eqTuple\ (n \in Scheme, A \in Typing\, n) \in rel\ (Tuple\, n\, A)$

$eqTuple\, n\, A \equiv \lambda\, x\, y \rightarrow \forall\, n\, \lambda\, i \rightarrow eqD\ (A\, i)\ (x\, i)\ (y\, i)$

Its definition involves the use of a universal quantifier over *Fin n* to express element-wise equality. Here the meaning to be read from it is "for all *i* in *Fin n*".

6.5.5 Relations

In this first attempt, we represent relations as boolean functions over tuples:

$Relation\ (n \in Scheme, I \in Typing\, n) \in Set$

$Relation\, n\, I \equiv Tuple\, n\, I \rightarrow Bool$

This gives us immediately the set operations over relations, by using the boolean operators in the usual way (union by boolean or, intersection by boolean and, difference by boolean and and complement).

$unionRel\ (n \in Scheme, I \in Typing\, n, R, S \in Relation\, n\, I) \in Relation\, n\, I$

$unionRel\, n\, I\, R\, S \equiv \lambda\, x \rightarrow R\, x \vee S\, x$

$interRel\ (n \in Scheme, I \in Typing\, n, R, S \in Relation\, n\, I) \in Relation\, n\, I$

$interRel\, n\, I\, R\, S \equiv \lambda\, x \rightarrow R\, x \wedge S\, x$

Also as in the preceding approach, selection is quite simple to define, by just taking the and of the (now boolean) selection predicate and the boolean function that represents the relation.

$selectRel\ (n \in Scheme, I \in Typing\, n, P \in pred\ (Tuple\, n\, I), R \in Relation\, n\, I) \in$
$Relation\, n\, I$

$selectRel\, n\, I\, P\, R \equiv \lambda\, x \rightarrow R\, x \wedge P\, x$

6.5.6 Projections

While schemes now have more structure, we still need something more to be able to decompose them in a way that allows us to define projection operations. We can try to use a solution analogous to the idea of using sum types in the preceding section. The difference is that we now use types of the form $\mathit{Fin}\ (m + n)$, with the idea that we can pick out the first part (of size m) and the second part (of size n). So the two projection functions will have types:

$$\begin{array}{l} \mathit{projRel1}\ (m, n \in \mathit{Scheme})(T \in \mathit{Typing}\ m)(T' \in \mathit{Typing}\ n) \\ \qquad\qquad (R \in \mathit{Rel}\ (m + n)\ (T + T')) \in \mathit{Rel}\ m\ T \end{array}$$

$$\begin{array}{l} \mathit{projRel2}\ (m, n \in \mathit{Scheme})(T \in \mathit{Typing}\ m)(T' \in \mathit{Typing}\ n) \\ \qquad\qquad (R \in \mathit{Rel}\ (m + n)\ (T + T')) \in \mathit{Rel}\ n\ T \end{array}$$

To project out the parts of a tuple we need, some properties of the finite sets are necessary. In particular, we need two functions that constitute the following isomorphism:

$$\mathit{Fin}\ m + \mathit{Fin}\ n \cong \mathit{Fin}\ (m + n)$$

Now we could try to define the projections by existential quantification over tuples, but the problem is that this doesn't give a decidable result, since we would have to search through a potentially infinite set.

As a consequence, it is natural to consider finite relations in this case, e.g. represented by lists, the relational operations represented as the corresponding list operations. We will address this approach in one of the following sections of this chapter.

Alternatively, one could imagine fixing the problem by attaching to the relations a cardinality, and trying to do existential quantification over the corresponding *Fin* set. Formally, this means that the extension of the boolean function is in one-to-one correspondence with a finite set $\mathit{Fin}\ n$. However, in general we can't know the exact cardinalities of relations coming from operations in this approach (in particular, think of union as an example for this problem). A possible solution will be presented after a final, brief mention of the operations with regard to this "exact cardinalities" approach.

6.5.7 Joins and dependencies

It is clear that the same problem regarding existential quantification will present itself when trying to define joins. Regarding functional dependencies, the problem is now a universal quantification over tuples, which again will not produce a decidable result.

6.5.8 A solution via approximate cardinalities

If we relax the requirement of having at all times the exact cardinality of each relation, it is possible to find a way around the problem of undecidable quantification when defining operations such as projections. We can represent a relation as a finite map, in the following way:

$$Rel\ (n \in Scheme)(T \in Typing\ n)(k \in Nat) \equiv Fin\ k \to Maybe(Tuple\ n\ T)$$

where as usual

$$Maybe\ (A \in Set) \equiv Ok\ (a \in A)\ \mid\ Fail$$

We remark that this representation can include several copies of the same tuple, in different arguments of the finite map. The reason for using a *Maybe* type on tuples is to have a cardinality that we can know at all times, and so be able to define relational operations that mention them. These cardinalities will be upper bounds on the actual number of tuples inside the relation, since the finite maps used for relations can have *Fail* values and so not be real tuples. The way in which this becomes an essential part of the definitions of the operations will be seen in the following explanations. The idea is that computing upper bounds on the cardinalities is easier than computing the exact cardinalities, and so will provide a possible solution to the problem of representing projections and joins.

By using the boolean-valued quantifiers provided by the standard Alfa library, we could define relational operations. (We refer the reader to the chapter on finite decidable allegories, where they are introduced.) In the case of union, what we need is actually a way of concatenating finite maps, to be used when defining union. The type of union would now look like this:

$$\begin{gathered} unionRel\ (n \in Scheme)(T \in Typing\ n)(k, k' \in Nat) \\ (R \in Rel\ n\ T\ k)(S \in Rel\ n\ T\ k') \in Rel\ n\ T\ (k + k') \end{gathered}$$

For intersection, the idea would be to iterate over the argument of the finite map for the first relation, and looking at the result of an existential quantifier over the finite map for the second relation. This would make use of a boolean conditional expression (if-then-else) applied to such quantifier, and giving either the original tuple if true, or *Fail* if false. That is, we "delete" (by changing them to *Fail*) all the real tuples in the first map that fail to appear in the second map. The typing of intersection would look like:

$$\begin{gathered} interRel\ (n \in Scheme)(T \in Typing\ n)(k, k' \in Nat) \\ (R \in Rel\ n\ T\ k)(S \in Rel\ n\ T\ k') \in Rel\ n\ T\ k \end{gathered}$$

For projections, we still keep the idea of viewing subschemes through types of the form $Fin\ (m+n)$. A projection could now be defined by iterating over the arguments of the finite map corresponding to the relation projected. Using the isomorphism mentioned in the previous discussion on projections, we can form the subtuples we need for each position in which real tuples are present (that is, when the finite map has value *Maybe t*). To give the type of the first projection, it would be:

$$\begin{array}{c} \mathit{projRel1}\ (m, n \in \mathit{Scheme})(T \in \mathit{Typing}\ m)(T' \in \mathit{Typing}\ n) \\ (k \in \mathit{Nat})(R \in \mathit{Rel}\ (m+n)\ (T+T')k) \in \mathit{Rel}\ m\, T k \end{array}$$

For joins, we need to perform what intuitively is a double iteration. For each tuple of the first relation, we can produce the finite map giving all the results of the fusion of that tuple with every tuple of the second relation. In case the fusion is not possible (when their common attributes have different values), we produce a *Fail* value. In this way, we know the result to have cardinality equal to the product of the joined relations. Needless to say, a lot of redundancy will appear as a result of this operation, since in most cases only a fraction of all tuples can actually be joined, and for every failed attempt we need to store such a *Fail* value. The typing of the join operation would look like this:

$$\begin{array}{c} \mathit{joinRel}\ (m, n, p \in \mathit{Scheme})(T_1 \in \mathit{Typing}\ m)(T_2 \in \mathit{Typing}\ n) \\ (T_3 \in \mathit{Typing}\ p)(k, k' \in \mathit{Nat})(R \in \mathit{Rel}\ (m+n)\ (T_1+T_2)k) \\ (S \in \mathit{Rel}\ (n+p)\ (T_2+T_3)k') \in \mathit{Rel}\ (m+n+p)\ (T_1+T_2+T_3)\ (k * k') \end{array}$$

It is also possible to define a notion of functional dependency, again based on getting subschemes through types of the form *Fin* $(m+n)$. The definition resorts to using a universal quantifier over the finite domain of the map for the relation that satisfies the functional dependency, following closely the set-theoretic definition. Regarding proofs of properties of database operations, we would need a great deal of properties about quantifiers over *Fin n* types. Some of these are provided by the standard library, but some are not. The proofs can also become a bit obscure and cumbersome, due not only to the recursions over such types that would make up the definitions, but also due to the use of boolean conditional expressions, for which we would need to establish a list of properties to carry proofs about the expressions in question.

6.5.9 Building up the cardinalities

In this section we have tried to solve the problems presented by choosing to represent finite-indexed, decidable relations, by means of particular finite mappings. The solution described in the preceding subsection starts with the cardinality that the result of a relational operation should have (as an upper bound), and then manipulates the mappings involved in a way that a new cardinality can be safely attached to the result.

Our approach in this section until now, and the particular solution in the preceding discussion of the problems encountered along the way, is based on handling relations as finite mappings in a very particular way: we start with the cardinality the result of a relational operation should (or will) have, and then manipulate mappings at the level of concatenating them.

A different approach could be to establish the cardinalities from the bottom of the definitions upwards. In this way, for defining intersection we would not start with the finite mapping corresponding to the first relation, and traverse through it erasing tuples that are not in the second relation by replacing them with *Fail* values as described above. What we would do is start with an empty finite map (that is, a map with domain *Fin* 0), and extend it to a domain one unit bigger each time we find a tuple that has to be in the result. While for some operations, like intersection, this way of proceeding would be particularly suitable, for others like projections it would be cumbersome. The trade-off is by no means clear, and the difficulty of carrying out proofs in this modified setting is probably just the same as in the preceding discussion.

Regardless of how we choose to build the cardinalities, a simpler and probably better solution is using lists for representing finite relations. The next section will present this solution.

6.6 Finite-indexed, list-based relations

6.6.1 Relations

This approach tries to solve the problems found in the preceding one, by using lists of tuples to represent relations. All the notions of attributes, domains, tuples and tuple equality are as in the previous section. The reader should be aware that repeated elements are allowed to appear in this representation of relations. Relations now become:

$Relation\ (n \in Nat, I \in Scheme\, n) \in Set$

$Relation\, n\, I \equiv List\ (Tuple\, n\, I)$

Set operations now don't follow immediately from the boolean or logical constants, but have to be implemented in terms of list operations. Union is simply the append of the two lists representing the relations. Intersection consists of filtering from one of the list those elements that are members of the other list. Also, selection is easy to define: it is just the usual filter operator for lists.

6.6.2 Projections

Using the same idea as in the preceding approach, and with the same types for the two projection functions, we can now fully define these operations. Since handling values of type $Fin(m+n)$ can be complicated, as we mentioned before we make use of an isomorphism into values of type $Plus(Fin\ m)(Fin\ n)$. (where *Plus* is the name of the sum type constructor in the Alfa library). This can be established from the Alfa library for finite sets, and we have a function

$splitFin\ (m, n \in Nat, k \in Fin\ (m + n)) \in Plus\ (Fin\, m)\ (Fin\, n)$

We also need a pair of functions: one that injects a value in $Fin\ m$ to a value of the bigger type $Fin\ (m + n)$, and another that injects a value in $Fin\ n$ to a value in $Fin\ m + Fin\ n$:

$raiseFin\ (m, n \in Nat, k \in Fin\, m) \in Fin\ (m + n)$

$raiseFin'\ (m, n \in Nat, k \in Fin\, n) \in Fin\ (m + n)$

The isomorphism is established by means of two lemmas, whose types follow:

$lemFinSplit\ (m, n \in Nat, i \in Fin\, m) \in$
$eqSumFin\, m\, n\ (splitFin\, m\, n\ (raiseFin\, m\, n\, i))\ (\mathbf{inl}\, i)$

$lemFinSplit'\ (m, n \in Nat, i \in Fin\, n) \in$
$eqSumFin\, m\, n\ (splitFin\, m\, n\ (raiseFin'\, m\, n\, i))\ (\mathbf{inr}\, i)$

Here we make use of the equality of sum types of two finite sets:

$eqSumFin\ (m, n \in Nat, x, y \in Plus\ (Fin\, m)\ (Fin\, n)) \in Prop$

which is substitutive, as will be indicated by:

$substEqSumFin\ (m, n \in Nat) \in Substitutive\ (Plus\ (Fin\, m)\ (Fin\, n))\ (eqSumFin\, m\, n)$

Sums of schemes

The actual function that takes two schemes (and their typings) and creates a new one corresponding to their intuitive sum is defined with the help of an auxiliary one.

$auxSchemes\ (m, n \in Nat, I \in Scheme\, m, J \in Scheme\, n, x \in Plus\ (Fin\, m)\ (Fin\, n)) \in$
$Datoid$

$auxSchemes\, m\, n\, I\, J\ (\mathbf{inl}\, x') \equiv I\, x'$
$auxSchemes\, m\, n\, I\, J\ (\mathbf{inr}\, x') \equiv J\, x'$

$addSchemes\ (m, n \in Nat, I \in Scheme\, m, J \in Scheme\, n) \in Scheme\ (m + n)$

$addSchemes\, m\, n\, I\, J \equiv \lambda\, k\ \rightarrow\ auxSchemes\, m\, n\, I\, J\ (splitFin\, m\, n\, k)$

Projection functions

First we define functions that project a tuple of size $m+n$ into the corresponding subtuples of sizes m and n. We remark that the definitions are by substitutivity of equality on sums of two finite sets:

$$\begin{array}{l} \mathit{projTuple1} \\ (m, n \in \mathit{Nat}, I \in \mathit{Scheme}\, m, J \in \mathit{Scheme}\, n, x \in \mathit{Tuple}\ (m+n)\ (\mathit{addSchemes}\, m\, n\, I\, J)) \in \\ \mathit{Tuple}\, m\, I \end{array}$$

$$\mathit{projTuple1}\, m\, n\, I\, J\, x \equiv \lambda\, i \ \rightarrow \begin{array}{l} \mathit{substEqSumFin}\, m\, n \\ \quad \lambda\, z \ \rightarrow |\, \mathit{auxSchemes}\, m\, n\, I\, J\, z\, | \\ \quad (\mathit{splitFin}\, m\, n\, (\mathit{raiseFin}\, m\, n\, i)) \\ \quad (\mathbf{inl}\, i) \\ \quad (\mathit{lemFinSplit}\, m\, n\, i) \\ \quad (x\, (\mathit{raiseFin}\, m\, n\, i)) \end{array}$$

$$\begin{array}{l} \mathit{projTuple2} \\ (m, n \in \mathit{Nat}, I \in \mathit{Scheme}\, m, J \in \mathit{Scheme}\, n, x \in \mathit{Tuple}\ (m+n)\ (\mathit{addSchemes}\, m\, n\, I\, J)) \in \\ \mathit{Tuple}\, n\, J \end{array}$$

$$\mathit{projTuple1}\, m\, n\, I\, J\, x \equiv \lambda\, i \ \rightarrow \begin{array}{l} \mathit{substEqSumFin}\, m\, n \\ \quad \lambda\, z \ \rightarrow |\, \mathit{auxSchemes}\, m\, n\, I\, J\, z\, | \\ \quad (\mathit{splitFin}\, m\, n\, (\mathit{raiseFin'}\, m\, n\, i)) \\ \quad (\mathbf{inr}\, i) \\ \quad (\mathit{lemFinSplit'}\, m\, n\, i) \\ \quad (x\, (\mathit{raiseFin'}\, m\, n\, i)) \end{array}$$

Now we can define the projection operations for a relation whose tuples are of size $m+n$. The definitions amount to a map over the list representing the relation, each tuple projected into the corresponding subtuple:

$$\begin{array}{l} \mathit{projRel1} \\ (m, n \in \mathit{Nat}, I \in \mathit{Scheme}\, m, J \in \mathit{Scheme}\, n, R \in \mathit{Relation}\ (m+n)\ (\mathit{addSchemes}\, m\, n\, I\, J)) \\ \in \mathit{Relation}\, m\, I \end{array}$$

$$\mathit{projRel1}\, m\, n\, I\, J\, R \equiv \begin{array}{l} \mathit{map}\ (\mathit{Tuple}\ (m+n)\ (\mathit{addSchemes}\, m\, n\, I\, J)) \\ (\mathit{Tuple}\, m\, I)\ (\mathit{projTuple1}\, m\, n\, I\, J)\ R \end{array}$$

$$\begin{array}{l} \mathit{projRel2} \\ (m, n \in \mathit{Nat}, I \in \mathit{Scheme}\, m, J \in \mathit{Scheme}\, n, R \in \mathit{Relation}\ (m+n)\ (\mathit{addSchemes}\, m\, n\, I\, J)) \\ \in \mathit{Relation}\, n\, J \end{array}$$

$$\mathit{projRel2}\, m\, n\, I\, J\, R \equiv \begin{array}{l} \mathit{map}\ (\mathit{Tuple}\ (m+n)\ (\mathit{addSchemes}\, m\, n\, I\, J)) \\ (\mathit{Tuple}\, n\, J)\ (\mathit{projTuple2}\, m\, n\, I\, J)\ R \end{array}$$

6.6.3 Joins

Joins can be defined too, by manipulating the lists corresponding to the two argument relations. The operation consists of comparing each pair of tuples, and producing a tuple in the result in case they match on their common attributes. The disassembling of the original tuples and the assembling of the resulting tuples are made by means of the isomorphism already mentioned.

$$
\mathit{joinRel}\left(\begin{array}{l} m, n, p \in \mathit{Nat}, I \in \mathit{Scheme}\, m, J \in \mathit{Scheme}\, n, K \in \mathit{Scheme}\, p, \\ R \in \mathit{Relation}\ (m+n)\ (\mathit{addSchemes}\, m\, n\, I\, J)\,, \\ S \in \mathit{Relation}\ (n+p)\ (\mathit{addSchemes}\, n\, p\, J\, K) \end{array}\right) \in
$$
$$
\mathit{Relation}\ ((m+n)+p)\ (\mathit{addSchemes}\ (m+n)\ p\ (\mathit{addSchemes}\, m\, n\, I\, J)\ K)
$$

$$\mathit{joinRel}\, m\, n\, p\, I\, J\, K\, []\, S \equiv []$$

$$\mathit{joinRel}\, m\, n\, p\, I\, J\, K\ (x : xs)\ [] \equiv []$$

$$
\begin{array}{l}
\mathit{joinRel}\, m\, n\, p\, I\, J\, K\ (x : xs)\ (x' : xs') \equiv \\
\quad \mathbf{let} \left[\begin{array}{l} \mathit{aux} \in \mathit{Relation}\ ((m+n)+p)\ (\mathit{addSchemes}\ (m+n)\ p\ (\mathit{addSchemes}\, m\, n\, I\, J)\ K) \\ \ \mathit{aux} \equiv \mathit{append}\, (\mathit{Tuple}\ ((m+n)+p) \\ \qquad (\mathit{addSchemes}\ (m+n)\ p\ (\mathit{addSchemes}\, m\, n\, I\, J)\ K)) \\ \qquad (\mathit{joinRel}\, m\, n\, p\, I\, J\, K\ (x : xs)\ xs')\ (\mathit{joinRel}\, m\, n\, p\, I\, J\, K\, xs\ (x' : xs')) \end{array}\right. \\
\quad \mathbf{in}\ \mathbf{if}\, \mathit{eqTuple}\, n\, J\ (\mathit{projTuple2}\, m\, n\, I\, J\, x)\ (\mathit{projTuple1}\, n\, p\, J\, K\, x') \\
\qquad \mathbf{then}\, \mathit{fuseTuples}\ (m+n)\ p\ (\mathit{addSchemes}\, m\, n\, I\, J)\ K\, x\ (\mathit{projTuple2}\, n\, p\, J\, K\, x') : \mathit{aux} \\
\qquad \mathbf{else}\, \mathit{aux}
\end{array}
$$

The definition makes use of an auxiliary function that puts together one of the tuples in the result, to be used when the two argument tuples coincide over the common part of their schemes.

$$
\mathit{fuseTuples}\, (m, n \in \mathit{Nat}, I \in \mathit{Scheme}\, m, J \in \mathit{Scheme}\, n, x \in \mathit{Tuple}\, m\, I, y \in \mathit{Tuple}\, n\, J) \in
$$
$$
\mathit{Tuple}\ (m+n)\ (\mathit{addSchemes}\, m\, n\, I\, J)
$$

$$
\mathit{fuseTuples}\, m\, n\, I\, J\, x\, y \equiv \lambda k \rightarrow \begin{array}{l} \mathit{elimPlus}\ (\mathit{Fin}\, m)\ (\mathit{Fin}\, n) \\ \quad \lambda a \rightarrow |\, \mathit{auxSchemes}\, m\, n\, I\, J\, a \,| \\ \quad x \\ \quad y \\ \quad (\mathit{splitFin}\, m\, n\, k) \end{array}
$$

6.6.4 Proving properties

Proving properties of the relational operations requires a good deal of the theory of lists. The definitions of the relational operations will make use of such list manipulations as filter, append, and structural recursion. Besides properties of these list operations, the already mentioned isomorphism for finite sets is used to deal with taking tuples apart. An additional difficulty is how to deal with two aspects of a list representation that are not part of the intuitive relational database model: repetitions and the order imposed on the tuples by the list. Clearly equality between relations should hold irrespective of such differences in the representation as a list, and this only makes it more difficult to establish the properties. The effort required is quite substantial, all things considered.

6.6.5 Dependencies

Similarly to the general indexed approach, we can define the notion of superkey, again using the analogy that $I + J$ sum types can be handled in a way like **Fin** $(m + n)$ types. The same issues and limitations from the past discussion apply in this analogous situation.

6.7 Databases via abstract finite sets

The final formalisation of databases in type theory is perhaps the most satisfactory. We shall introduce an "abstract" notion of database which can be instantiated to any of several "concrete" implementations, for instance the various kinds of hash tables or balanced search trees that are actually used in implementations of database management systems. An important point to notice is that many of the useful implementations of the set data type are based on comparing elements not just for equality but for total ordering. With this in mind, we will define an *ordered* data type to capture such behaviour, and require that the abstract set data type has as elements members of such an ordered data type.

The idea is to define a database as an abstract finite set of tuples over a scheme. We will concentrate on tuples over schemes represented by *Fin n* types, as in previous sections. This means we assume that the attributes are the numbers $0, 1, \ldots, n - 1$ that inhabit the set *Fin n*. As mentioned in previous sections, if it is desired to have attribute names we can provide a one-to-one mapping between *Fin n* and strings.

An important remark that the reader should keep in mind during the following is that the actual shape of the abstract data type, and the idioms of it (that is, the choice of names, axioms, parameter order and such) follow those used in the work of Filliatre and Letouzey [29] on implementations of finite sets in Coq. We have simplified in parts, and adapted many details of how things are expressed in Coq to the way they can be expressed in Alfa/Agda, of course.

6.7.1 Ordered types

To formalise a completely general idea of a type with an order over it, we start by defining an auxiliary type. The idea is that given operations of comparison for equality and strict less-than order over a data type, we will always have an element in this algebraic type, where the constructor will say which of the three possibilities $x < y$, $x = y$, $y < x$ actually holds, and at the same time provides us with the actual proof of this fact.

$$Compare \begin{pmatrix} X \in Set \\ lt \in X \to X \to Prop \\ eq \in X \to X \to Prop \\ x \in X \\ y \in X \end{pmatrix} \in Set$$

$$Compare\, X\, lt\, eq\, x\, y \equiv \begin{array}{l} \textbf{data} \\ \quad \textbf{Lt}\,(h \in lt\,x\,y) \\ \quad \textbf{Eq}\,(h \in eq\,x\,y) \\ \quad \textbf{Gt}\,(h \in lt\,y\,x) \end{array}$$

The actual ordered type definition follows, in the shape of a signature. An ordered type is a set equipped with both an equality and a less-than relations, the equality being reflexive, symmetric and transitive, while the less-than is transitive. An axiom specifies that less-than behaves intuitively right with respect to the equality. Finally, an operation using the previous auxiliary comparison algebraic type produces the result of comparing two elements of the type, in the already mentioned sense of producing a constructor marking the result and containing a proof of the result In this way, we obtain in a roundabout way a decidable operation of comparison, which satisfies a kind of trichotomy law. The last component of the signature is an axiom that makes less-than substitutive with respect to the equality of the ordered type.

$$OrderedType \in Type$$

$$OrderedType \equiv \begin{array}{l} \textbf{sig} \\ \quad t \in Set \\ \quad eq \in t \to t \to Prop \\ \quad lt \in t \to t \to Prop \\ \quad eqRefl \in Reflexive\,t\,eq \\ \quad eqSym \in Symmetrical\,t\,eq \\ \quad eqTran \in Transitive\,t\,eq \\ \quad ltTran \in Transitive\,t\,lt \\ \quad ltNotEq \in (x, y \in t) \to lt\,x\,y \to \neg\,(eq\,x\,y) \\ \quad compare \in (x, y \in t) \to Compare\,t\,lt\,eq\,x\,y \\ \quad ltSubst \in (x, x', y, y' \in t) \to eq\,x\,x' \to eq\,y\,y' \to lt\,x\,y \to lt\,x'\,y' \end{array}$$

The following two lemmas establish some basic properties of the less-than relation of an ordered type and will be useful later:

$$lemEqNotLt \begin{pmatrix} A \in OrderedType \\ x \in A.t \\ y \in A.t \\ h \in A.eq\,x\,y \end{pmatrix} \in \neg\ (A.lt\,x\,y)$$

$$lemEqNotLt\,A\,x\,y\,h \equiv \lambda\,h' \ \rightarrow\ A.ltNotEq\,x\,y\,h'\,h$$

$$lemLtNotGt \begin{pmatrix} A \in OrderedType \\ x \in A.t \\ y \in A.t \\ h \in A.lt\,x\,y \end{pmatrix} \in \neg\ (A.lt\,y\,x)$$

$$lemLtNotGt\,A\,x\,y\,h \equiv \lambda\,h' \ \rightarrow\ lemEqNotLt\,A\,x\,x\ (A.eqRefl\,x)\ (A.ltTran\,x\,y\,x\,h\,h')$$

Finally, it is useful to have an elimination constant for the *Compare* algebraic data type. This will allow to perform definitions by case analysis on the result of a compare operation, while still being able to prove things about such definitions.

$$elimCompare \begin{pmatrix} X \in Set \\ lt \in X \ \rightarrow\ X \ \rightarrow\ Prop \\ eq \in X \ \rightarrow\ X \ \rightarrow\ Prop \\ x \in X \\ y \in X \\ P \in Pred\ (Compare\,X\,lt\,eq\,x\,y) \\ hL \in (h\ \in\ lt\,x\,y) \rightarrow\ P\ (\mathbf{Lt}\,h) \\ hE \in (h\ \in\ eq\,x\,y) \rightarrow\ P\ (\mathbf{Eq}\,h) \\ hG \in (h\ \in\ lt\,y\,x) \rightarrow\ P\ (\mathbf{Gt}\,h) \\ c \in Compare\,X\,lt\,eq\,x\,y \end{pmatrix} \in P\,c$$

$$elimCompare\,X\,lt\,eq\,x\,y\,P\,hL\,hE\,hG\ (\mathbf{Lt}\,h) \equiv hL\,h$$
$$elimCompare\,X\,lt\,eq\,x\,y\,P\,hL\,hE\,hG\ (\mathbf{Eq}\,h) \equiv hE\,h$$
$$elimCompare\,X\,lt\,eq\,x\,y\,P\,hL\,hE\,hG\ (\mathbf{Gt}\,h) \equiv hG\,h$$

6.7.2 Abstract finite sets

The definition is formalized in two parts for convenience: a signature *ASet* taking an ordered data type and providing all the operations of an abstract finite set data type, and a second signature *ASetSpec* (with the same kind of parameter) that provides all the axioms that must be satisfied by implementations of the abstract finite set type.

$$ASet\ (X\ \in\ OrderedType) \in Type$$

$$
\begin{array}{l}
ASet\ X \equiv \\
\quad \mathbf{sig} \\
\qquad t \in Set \\
\qquad empty \in t \\
\qquad isEmpty \in t \to Bool \\
\qquad mem \in X.t \to t \to Bool \\
\qquad add \in X.t \to t \to t \\
\qquad singleton \in X.t \to t \\
\qquad remove \in X.t \to t \to t \\
\qquad union \in t \to t \to t \\
\qquad inter \in t \to t \to t \\
\qquad diff \in t \to t \to t \\
\qquad eq \in t \to t \to Prop \\
\qquad lt \in t \to t \to Prop \\
\qquad fold \in (A \in Set) \to (X.t \to A \to A) \to t \to A \to A \\
\qquad forAll \in (X.t \to Bool) \to t \to Bool \\
\qquad exists \in (X.t \to Bool) \to t \to Bool \\
\qquad filter \in (X.t \to Bool) \to t \to t \\
\qquad partition \in (X.t \to Bool) \to t \to Times\ t\ t \\
\qquad cardinal \in t \to Nat \\
\qquad elements \in t \to List\ X.t \\
\qquad minElt \in t \to Maybe\ X.t \\
\qquad maxElt \in t \to Maybe\ X.t \\
\qquad choose \in t \to Maybe\ X.t
\end{array}
$$

The names of the operations should be self-evident in many cases. We remark that besides the usual operations for such a data type that reflect the naive set theoretical ones, there are a few more that are useful for the manipulation of the data type. In particular, a fold operation is provided that intuitively behaves as the list fold for sets. Universal and existential decidable quantifiers, filtering and partitioning according to a decidable predicate, finding minimum and maximum elements and the set cardinality are other such special operations. An operation to choose an arbitrary element of the abstract set is the last one provided in the signature, though given the nature of Alfa/Agda's semantics, we won't get a nondeterministic implementation to be provided as part of an instance of a structure for this signature.

For some operations, a result in *Maybe X.t* is used, this is the usual type with two constructors, *Ok x* and *Fail*. The equality on this type will be needed later to specify the laws of abstract finite sets, and it has type:

$$
eqMaybe \left(\begin{array}{l} A \in Setoid \\ x \in Maybe\ (|\,A\,|) \\ y \in Maybe\ (|\,A\,|) \end{array} \right) \in Prop
$$

The axioms (laws) for any desired implementation of the abstract data type are provided, as already mentioned, by another signature:

$\mathit{ASetSpec}\ (X \in \mathit{OrderedType}) \in \mathit{Type}$

$\mathit{ASetSpec}\ X \equiv$
sig
$S \in \mathit{ASet}\ X$
open S **use** $t, \mathit{empty}, \mathit{isEmpty}, \mathit{mem}, \mathit{add}, \mathit{remove}, \mathit{singleton}, \mathit{union}, \mathit{inter}, \mathit{diff}, \mathit{eq}, \mathit{lt},$
$\quad \mathit{fold}, \mathit{filter}, \mathit{forAll}, \mathit{exists}, \mathit{partition}, \mathit{cardinal}, \mathit{elements}, \mathit{minElt}, \mathit{maxElt}, \mathit{choose}$
$\mathit{axIn} \in (x, y \in X.t, s \in t) \to X.\mathit{eq}\,x\,y \to (|\,\mathit{mem}\,x\,s\,|) \to |\,\mathit{mem}\,y\,s\,|$
$\mathit{axEqRefl} \in \mathit{Reflexive}\ t\ \mathit{eq}$
$\mathit{axEqSym} \in \mathit{Symmetrical}\ t\ \mathit{eq}$
$\mathit{axEqTrans} \in \mathit{Transitive}\ t\ \mathit{eq}$
$\mathit{axLtTrans} \in \mathit{Transitive}\ t\ \mathit{lt}$
$\mathit{axLtNotEq} \in (s, s' \in t) \to \mathit{lt}\,s\,s' \to \neg\,(\mathit{eq}\,s\,s')$
$\mathit{axEmpty1} \in |\,\mathit{isEmpty}\ \mathit{empty}\,|$
$\mathit{axEmpty2} \in (s \in t) \to (|\,\mathit{isEmpty}\,s\,|) \leftrightarrow (x \in X.t) \to \neg\,(|\,\mathit{mem}\,x\,s\,|)$
$\mathit{axEqual} \in (s, s' \in t) \to \mathit{eq}\,s\,s' \leftrightarrow (x \in X.t) \to ((|\,\mathit{mem}\,x\,s\,|) \leftrightarrow (|\,\mathit{mem}\,x\,s'\,|))$
$\mathit{axSubset} \in (s, s' \in t) \to \mathit{lt}\,s\,s' \leftrightarrow (x \in X.t) \to ((|\,\mathit{mem}\,x\,s\,|) \supset (|\,\mathit{mem}\,x\,s'\,|))$
$\mathit{axAdd1} \in (x, y \in X.t, s \in t) \to X.\mathit{eq}\,x\,y \supset (|\,\mathit{mem}\,x\,(\mathit{add}\,y\,s)\,|)$
$\mathit{axAdd2} \in (x, y \in X.t, s \in t) \to (|\,\mathit{mem}\,x\,s\,|) \supset (|\,\mathit{mem}\,x\,(\mathit{add}\,y\,s)\,|)$
$\mathit{axAdd3} \in (x, y \in X.t, s \in t) \to \neg\,(X.\mathit{eq}\,x\,y) \to (|\,\mathit{mem}\,x\,(\mathit{add}\,y\,s)\,|) \supset (|\,\mathit{mem}\,x\,s\,|)$
$\mathit{axRemove1} \in (x, y \in X.t, s \in t) \to X.\mathit{eq}\,x\,y \supset \neg\,(|\,\mathit{mem}\,x\,(\mathit{remove}\,y\,s)\,|)$
$\mathit{axRemove2} \in (x, y \in X.t, s \in t) \to \neg\,(X.\mathit{eq}\,x\,y) \to$
$\quad (|\,\mathit{mem}\,x\,s\,|) \supset (|\,\mathit{mem}\,x\,(\mathit{remove}\,y\,s)\,|)$
$\mathit{axRemove3} \in (x, y \in X.t, s \in t) \to (|\,\mathit{mem}\,x\,(\mathit{remove}\,y\,s)\,|) \supset (|\,\mathit{mem}\,x\,s\,|)$
$\mathit{axSingleton} \in (x, y \in X.t) \to (|\,\mathit{mem}\,x\,(\mathit{singleton}\,y)\,|) \leftrightarrow X.\mathit{eq}\,x\,y$
$\mathit{axUnion1} \in (x \in X.t, s, s' \in t) \to$
$\quad (|\,\mathit{mem}\,x\,(\mathit{union}\,s\,s')\,|) \supset ((|\,\mathit{mem}\,x\,s\,|) \vee (|\,\mathit{mem}\,x\,s'\,|))$
$\mathit{axUnion2} \in (x \in X.t, s, s' \in t) \to (|\,\mathit{mem}\,x\,s\,|) \supset (|\,\mathit{mem}\,x\,(\mathit{union}\,s\,s')\,|)$
$\mathit{axUnion3} \in (x \in X.t, s, s' \in t) \to (|\,\mathit{mem}\,x\,s'\,|) \supset (|\,\mathit{mem}\,x\,(\mathit{union}\,s\,s')\,|)$
$\mathit{axInter1} \in (x \in X.t, s, s' \in t) \to (|\,\mathit{mem}\,x\,(\mathit{inter}\,s\,s')\,|) \supset (|\,\mathit{mem}\,x\,s\,|)$
$\mathit{axInter2} \in (x \in X.t, s, s' \in t) \to (|\,\mathit{mem}\,x\,(\mathit{inter}\,s\,s')\,|) \supset (|\,\mathit{mem}\,x\,s'\,|)$
$\mathit{axInter3} \in (x \in X.t, s, s' \in t) \to$
$\quad (|\,\mathit{mem}\,x\,s\,|) \to (|\,\mathit{mem}\,x\,s'\,|) \to |\,\mathit{mem}\,x\,(\mathit{inter}\,s\,s')\,|$
$\mathit{axDiff1} \in (x \in X.t, s, s' \in t) \to (|\,\mathit{mem}\,x\,(\mathit{diff}\,s\,s')\,|) \supset (|\,\mathit{mem}\,x\,s\,|)$
$\mathit{axDiff2} \in (x \in X.t, s, s' \in t) \to (|\,\mathit{mem}\,x\,(\mathit{diff}\,s\,s')\,|) \supset \neg\,(|\,\mathit{mem}\,x\,s'\,|)$
$\mathit{axDiff3} \in (x \in X.t, s, s' \in t) \to$
$\quad (|\,\mathit{mem}\,x\,s\,|) \to \neg\,(|\,\mathit{mem}\,x\,s'\,|) \to |\,\mathit{mem}\,x\,(\mathit{diff}\,s\,s')\,|$
$\mathit{axFold} \in (A \in \mathit{Setoid}, a \in |\,A\,|, f \in X.t \to (|\,A\,|) \to |\,A\,|, s \in t) \to$
$\quad \exists\, l \in \mathit{List}\,X.t\,.\ \mathit{Unique}\ (\mathit{SetoidFromOrder}\ X)\ l \quad \&$
$\qquad ((\forall\, x \in X.t\,.\ (|\,\mathit{mem}\,x\,s\,|) \leftrightarrow \mathit{inList}\ (\mathit{SetoidFromOrder}\ X)\ x\,l) \quad \&$
$\qquad\quad == A\ (\mathit{fold}\ (|\,A\,|)\ f\,s\,a)\ (\mathit{foldr}\ X.t\ (|\,A\,|)\ f\,a\,l))$

$$
\begin{array}{l}
\mathit{axCardinal} \in (s \in t) \to \exists\, l \in \mathit{List}\, X.t\,.\ \mathit{Unique}\ (\mathit{SetoidFromOrder}\, X)\ l \quad \& \\
\qquad ((\forall\, x \in X.t\,.\ (|\, \mathit{mem}\, x\, s\,|) \leftrightarrow \\
\qquad\quad \mathit{inList}\ (\mathit{SetoidFromOrder}\, X)\, x\, l)\ \&\ (|\, \mathit{cardinal}\, s == \mathit{length}\, X.t\, l\,|)) \\
\mathit{axFilter1} \in (f \in X.t \to \mathit{Bool}) \to \mathit{compatBool}\ (\mathit{SetoidFromOrder}\, X)\ f \to \\
\qquad (x \in X.t, s \in t) \to (|\, \mathit{mem}\, x\ (\mathit{filter}\, f\, s)\ |) \to |\, \mathit{mem}\, x\, s\,| \\
\mathit{axFilter2} \in (f \in X.t \to \mathit{Bool}) \to \mathit{compatBool}\ (\mathit{SetoidFromOrder}\, X)\ f \to \\
\qquad (x \in X.t, s \in t) \to (|\, \mathit{mem}\, x\ (\mathit{filter}\, f\, s)\ |) \to |\, f\, x\,| \\
\mathit{axFilter3} \in (f \in X.t \to \mathit{Bool}) \to \mathit{compatBool}\ (\mathit{SetoidFromOrder}\, X)\ f \to \\
\qquad (x \in X.t, s \in t) \to (|\, \mathit{mem}\, x\, s\,|) \to (|\, f\, x\,|) \to |\, \mathit{mem}\, x\ (\mathit{filter}\, f\, s)\ | \\
\mathit{axForall1} \in (f \in X.t \to \mathit{Bool}, s \in t) \to \mathit{compatBool}\ (\mathit{SetoidFromOrder}\, X)\ f \to \\
\qquad (\forall\, x \in X.t\,.\ (|\, \mathit{mem}\, x\, s\,|) \supset (|\, f\, x\,|)) \to |\, \mathit{forAll}\, f\, s\,| \\
\mathit{axForall2} \in (f \in X.t \to \mathit{Bool}, s \in t) \to \\
\qquad \mathit{compatBool}\ (\mathit{SetoidFromOrder}\, X)\ f \to (|\, \mathit{forAll}\, f\, s\,|) \to \\
\qquad \forall\, x \in X.t\,.\ (|\, \mathit{mem}\, x\, s\,|) \supset (|\, f\, x\,|) \\
\mathit{axExists1} \in (f \in X.t \to \mathit{Bool}, s \in t) \to \\
\qquad \mathit{compatBool}\ (\mathit{SetoidFromOrder}\, X)\ f \to \\
\qquad (\exists\, x \in X.t\,.\ (|\, \mathit{mem}\, x\, s\,|)\ \&\ (|\, f\, x\,|)) \to |\, \mathit{exists}\, f\, s\,| \\
\mathit{axExists2} \in (f \in X.t \to \mathit{Bool}, s \in t) \to \\
\qquad \mathit{compatBool}\ (\mathit{SetoidFromOrder}\, X)\ f \to (|\, \mathit{exists}\, f\, s\,|) \to \\
\qquad \exists\, x \in X.t\,.\ (|\, \mathit{mem}\, x\, s\,|)\ \&\ (|\, f\, x\,|) \\
\mathit{axElems1} \in (x \in X.t, s \in t) \to (|\, \mathit{mem}\, x\, s\,|) \to \\
\qquad \mathit{inList}\ (\mathit{SetoidFromOrder}\, X)\ x\ (\mathit{elements}\, s) \\
\mathit{axElems2} \in (x \in X.t, s \in t) \to \\
\qquad \mathit{inList}\ (\mathit{SetoidFromOrder}\, X)\ x\ (\mathit{elements}\, s) \to |\, \mathit{mem}\, x\, s\,| \\
\mathit{axElems3} \in (s \in t) \to \mathit{sortedList}\, X.t\, X.lt\ (\mathit{elements}\, s) \\
\mathit{axMin1} \in (s \in t, x \in X.t) \to \\
\qquad \mathit{eqMaybe}\ (\mathit{SetoidFromOrder}\, X)\ (\mathit{minElt}\, s)\ (\mathbf{Ok}\, x) \to |\, \mathit{mem}\, x\, s\,| \\
\mathit{axMin2} \in (s \in t, x, y \in X.t) \to \mathit{eqMaybe}\ (\mathit{SetoidFromOrder}\, X)\ (\mathit{minElt}\, s)\ (\mathbf{Ok}\, x) \to \\
\quad (|\, \mathit{mem}\, y\, s\,|) \to \neg\ (X.lt\, y\, x) \\
\mathit{axMin3} \in (s \in t) \to \mathit{eqMaybe}\ (\mathit{SetoidFromOrder}\, X)\ (\mathit{minElt}\, s)\ \mathbf{Fail} \to |\, \mathit{isEmpty}\, s\,| \\
\mathit{axMax1} \in (s \in t, x \in X.t) \to \\
\qquad \mathit{eqMaybe}\ (\mathit{SetoidFromOrder}\, X)\ (\mathit{maxElt}\, s)\ (\mathbf{Ok}\, x) \to |\, \mathit{mem}\, x\, s\,| \\
\mathit{axMax2} \in (s \in t, x, y \in X.t) \to \\
\qquad \mathit{eqMaybe}\ (\mathit{SetoidFromOrder}\, X)\ (\mathit{maxElt}\, s)\ (\mathbf{Ok}\, x) \to \\
\qquad (|\, \mathit{mem}\, y\, s\,|) \to \neg\ (X.lt\, x\, y) \\
\mathit{axMax3} \in (s \in t) \to \mathit{eqMaybe}\ (\mathit{SetoidFromOrder}\, X)\ (\mathit{maxElt}\, s)\ \mathbf{Fail} \to |\, \mathit{isEmpty}\, s\,| \\
\mathit{axChoose1} \in (s \in t, x \in X.t) \to \\
\qquad \mathit{eqMaybe}\ (\mathit{SetoidFromOrder}\, X)\ (\mathit{choose}\, s)\ (\mathbf{Ok}\, x) \to |\, \mathit{mem}\, x\, s\,| \\
\mathit{axChoose2} \in (s \in t) \to \\
\qquad \mathit{eqMaybe}\ (\mathit{SetoidFromOrder}\, X)\ (\mathit{choose}\, s)\ \mathbf{Fail} \to |\, \mathit{isEmpty}\, s\,| \\
\mathit{axPartition1} \in (s \in t, f \in X.t \to \mathit{Bool}) \to \mathit{compatBool}\ (\mathit{SetoidFromOrder}\, X)\ f \to \\
\quad \mathit{eq}\ (\mathit{partition}\, f\, s)\, .\mathit{fst}\ (\mathit{filter}\, f\, s) \\
\mathit{axPartition2} \in (s \in t, f \in X.t \to \mathit{Bool}) \to \mathit{compatBool}\ (\mathit{SetoidFromOrder}\, X)\ f \to \\
\quad \mathit{eq}\ (\mathit{partition}\, f\, s)\, .\mathit{snd}\ (\mathit{filter}\ \lambda\, x \to \neg\ (f\, x)\ s)
\end{array}
$$

A few auxiliary definitions and data types are used in the preceding axioms. For lists, using the definition of the standard Alfa libraries, we provide operations for length, membership and tests for sortedness and nonrepetition of elements:

$$length \begin{pmatrix} A \in Set \\ l \in List\, A \end{pmatrix} \in Nat$$

$$length\, A\, [\,] \equiv \mathbf{zer}$$

$$length\, A\ (x\ :\ xs) \equiv \mathbf{suc}\ (length\, A\, xs)$$

$$inList \begin{pmatrix} A \in Setoid \\ x \in |\, A\,| \\ l \in List\ (|\, A\,|) \end{pmatrix} \in Prop$$

$$inList\, A\, x\, [\,] \equiv \bot$$

$$inList\, A\, x\ (x'\ :\ xs) \equiv ==\ A\, x\, x' \vee inList\, A\, x\, xs$$

$$sortedList \begin{pmatrix} A \in Set \\ lt \in Rel\, A \\ l \in List\, A \end{pmatrix} \in Prop$$

$$sortedList\, A\, lt\, [\,] \equiv T$$

$$sortedList\, A\, lt\ (x\ :\ [\,]) \equiv T$$

$$sortedList\, A\, lt\ (x\ :\ (x'\ :\ xs')) \equiv lt\, x\, x'\ \&\ sortedList\, A\, lt\, xs$$

$$Unique\ (A\ \in\ Setoid) \in Pred\ (List\ (|\, A\,|))$$

$$Unique\, A \equiv \lambda\, l\ \rightarrow \quad \begin{array}{l} \textbf{case}\ l\ \textbf{of} \\ \quad [\,]\ \rightarrow\ T \\ \quad x\ :\ xs\ \rightarrow\ Unique\, A\, xs\ \&\ \neg\ (inList\, A\, x\, xs) \end{array}$$

The *foldr* used in *axFold* is the standard fold-right combinator for lists, and is provided by the Alfa libraries.

A repackaging operation is provided for forming the implicit setoid that is present as part of any ordered type, when you consider only the equality operation:

$$SetoidFromOrder\ (X\ \in\ OrderedType) \in Setoid$$

$$\begin{array}{l} SetoidFromOrder\, X \equiv \\ \quad \textbf{struct} \\ \quad \left[\begin{array}{l} Elem \equiv X.t \\ Equal \equiv X.eq \\ ref \equiv X.eqRefl \\ sym \equiv \lambda\, x1\, x2\, h\ \rightarrow\ X.eqSubst\ \lambda\, x\ \rightarrow\ X.eq\, x\, x1\, x1\, x2\, h\ (X.eqRefl\, x1) \\ tran \equiv \lambda\, x1\, x2\, x3\, h\, h'\ \rightarrow\ X.eqSubst\ (X.eq\, x1)\ x2\, x3\, h'\, h \end{array} \right. \end{array}$$

Also, for several laws involving decidable (boolean) predicates, we define what it means for the predicate to be compatible with the equality relation of a setoid:

$$compatBool \left(\begin{array}{l} A \in Setoid \\ f \in (|\, A \,|) \rightarrow Bool \end{array} \right) \in Prop$$

$$compatBool\, A\, f \equiv (x, y \in |\, A \,|) \rightarrow == A\, x\, y \rightarrow eqBool\ (f\, x)\ (f\, y)$$

Here *eqBool* is the equality for the algebraic datatype *Bool*.

6.7.3 An implementation through basic collections

A different name for a data type that captures the concept of finite sets over a base data type is *collection* (sometimes also referred to as *bulk types*). In what follows, we will use this name to refer to a particular kind of implementation of the abstract finite set signature, one that uses lists without repeated elements. This takes some inspiration from the work carried out by Rajagopalan in NuPRL [61]. That work is concerned with establishing an abstract algebra of collections that is suitable for embedding semantic data models (besides the relational model). This is a different approach from the one we attempt here, but the common insight of both efforts is that collections are a useful way of formalizing database theory. In particular, both abstract and concrete notions of collections seem very well suited to that kind of work.

As a matter of fact, we should note that the development of a formalisation of collections in this sense in Alfa preceded our abstract finite set interface approach. Here we will present our formalisation of collections and show that it is in fact an implementation of that abstract interface.

A generic collection will be a list that satisfies some property of interest, and we make the notion of collection parametric with respect to this property:

$$GenColl\ (A \in Set, P \in Pred\ (List\, A)) \in Set$$

$$GenColl\, A\, P \equiv \quad \begin{array}{l} \textbf{sig} \\ \quad elems \in List\, A \\ \quad ppty \in P\, elems \end{array}$$

While many such interesting properties could be used (for instance, being in ascending order), for our purposes we will use a P that verifies that the list doesn't contain repeated elements. Since the proof of such a property requires comparing elements of the list for equality, we work with a base type for the collection that is a setoid.

$$Coll\ (A \in Setoid) \in Set$$

$Coll\, A \equiv GenColl\ (|\, A\, |)\ (noDupls\, A)$

Here *noDupls* is the predicate that holds over lists that don't have duplicate elements (called *noDupls*), and makes use of the membership operation over lists (here denoted as *member*). It is important to remark that the shape of *Coll* means that each time we operate on a *Coll* we need to prove that the list resulting from the corresponding manipulation will not have repetitions.

However, the requirement that the base type is a setoid is not enough, as we would like this to be a datatype that provides decidable and computable operations. For instance, the membership test cannot just give a result in *Prop*. An alternative is to require that the equality on A is decidable in the sense that P is decidable if and only if $P \vee \neg P$ holds. This notion is provided by the Alfa library under the name *Decidable*, and a corresponding *DecidableRel* is also provided for relations.

Another alternative is to require $A \in Datoid$, as used in other approaches in the preceding sections. For the purposes of establishing a theory of collections, whose properties we can use for proving things about databases defined in terms of them, the first alternative seems easier. It will also offer an example of using decidable properties of a different shape, something that we haven't done before in the present thesis.

Decidable propositions

We establish that combining decidable propositions in the above sense using any of the logical connectives (including constants) produces decidable propositions. This formalization is our own, since the Alfa library lacks the definitions and proofs necessary for dealing with decidability in this sense.

$decTriv\ \in Decidable\, \mathsf{T}$

$decTriv \equiv \mathbf{inl}\ \mathbf{tt}$

$decAbs\ \in Decidable\, \perp$

$decAbs \equiv \mathbf{inr}\ \lambda h\ \rightarrow\ h$

$decNot\ (P\ \in\ Prop, dP\ \in\ Decidable\, P) \in Decidable\ (\neg\, P)$

$decNot\, P\ \ (\mathbf{inl}\ isP) \equiv \mathbf{inr}\ \lambda h\ \rightarrow\ h\ isP$
$decNot\, P\ \ (\mathbf{inr}\ notP) \equiv \mathbf{inl}\ notP$

$decAnd\ (P, Q\ \in\ Prop, dP\ \in\ Decidable\, P, dQ\ \in\ Decidable\, Q) \in Decidable\ (P \,\&\, Q)$

$$decAnd\,P\,Q\ (\mathbf{inl}\,isP)\ (\mathbf{inl}\,isQ) \equiv \mathbf{inl}\left(\mathbf{struct}\left\{\left[\begin{array}{l} fst \equiv isP \\ snd \equiv isQ \end{array}\right\}\right)\right.$$

$$decAnd\,P\,Q\ (\mathbf{inl}\,isP)\ (\mathbf{inr}\,notQ) \equiv \mathbf{inr}\,\lambda\,h\ \rightarrow\ notQ\,h.snd$$

$$decAnd\,P\,Q\ (\mathbf{inr}\,notP)\ dQ \equiv \mathbf{inr}\,\lambda\,h\ \rightarrow\ notP\,h.fst$$

$$decOr\,(P, Q\ \in\ Prop, dP\ \in\ Decidable\,P, dQ\ \in\ Decidable\,Q) \in Decidable\ (P\ \vee\ Q)$$

$$decOr\,P\,Q\ (\mathbf{inl}\,isP)\ dQ \equiv \mathbf{inl}\ (\mathbf{inl}\,isP)$$

$$decOr\,P\,Q\ (\mathbf{inr}\,notP)\ (\mathbf{inl}\,isQ) \equiv \mathbf{inl}\ (\mathbf{inr}\,isQ)$$

$$decOr\,P\,Q\ (\mathbf{inr}\,notP)\ (\mathbf{inr}\,notQ) \equiv \mathbf{inr}\,\lambda\,h\ \rightarrow\ elimOr\,P\,Q\,\bot\,notP\,notQ\,h$$

$$decImpl\,(P, Q\ \in\ Prop, dP\ \in\ Decidable\,P, dQ\ \in\ Decidable\,Q) \in Decidable\ (P\ \supset\ Q)$$

$$decImpl\,P\,Q\ (\mathbf{inl}\,isP)\ (\mathbf{inl}\,isQ) \equiv \mathbf{inl}\,\lambda\,h\ \rightarrow\ isQ$$

$$decImpl\,P\,Q\ (\mathbf{inl}\,isP)\ (\mathbf{inr}\,notQ) \equiv \mathbf{inr}\,\lambda\,h\ \rightarrow\ notQ\ (h\,isP)$$

$$decImpl\,P\,Q\ (\mathbf{inr}\,notP)\ dQ \equiv \mathbf{inl}\,\lambda\,h\ \rightarrow\ elimAbsurd\,Q\ (notP\,h)$$

$$decIff\,(P, Q\ \in\ Prop, dP\ \in\ Decidable\,P, dQ\ \in\ Decidable\,Q) \in Decidable\ (P\ \leftrightarrow\ Q)$$

$$decIff\,P\,Q\,dP\,dQ \equiv decAnd\ (P\ \supset\ Q)\ (Q\ \supset\ P)\ (decImpl\,P\,Q\,dP\,dQ)\ (decImpl\,Q\,P\,dQ\,dP)$$

We will make much use of a form of if-then-else definition for this kind of decidable proposition. The idea is that we examine a proof of the form $P \vee \neg P$, by looking at the constructor tagging it (it would be either an **inl** or an **inr**). Each case then gives rise to the two branches of the conditional definition.

$$ifD\ (A\ \in\ Set, P\ \in\ Prop, c\ \in\ Decidable\,P, t\ \in\ P\ \rightarrow\ A, e\ \in\ \neg\,P\ \rightarrow\ A) \in A$$

$$ifD\,A\,P\ (\mathbf{inl}\,isP)\ t\,e \equiv t\,isP$$

$$ifD\,A\,P\ (\mathbf{inr}\,notP)\ t\,e \equiv e\,notP$$

$$ifD'\ (A\ \in\ Type, P\ \in\ Prop, c\ \in\ Decidable\,P, t\ \in\ P\ \rightarrow\ A, e\ \in\ \neg\,P\ \rightarrow\ A) \in A$$

$$ifD'\,A\,P\ (\mathbf{inl}\,isP)\ t\,e \equiv t\,isP$$

$$ifD'\,A\,P\ (\mathbf{inr}\,notP)\ t\,e \equiv e\,notP$$

For proving properties of operations defined making use of this particular form of if-then-else, we need an elimination constant. The definition follows:

$$\begin{array}{l} undistIfD \\ \quad (A\ \in\ Set, Q\ \in\ Pred\,A, P\ \in\ Prop, c\ \in\ Decidable\,P, t\ \in\ P\ \rightarrow\ A \\ \qquad e\ \in\ \neg\,P\ \rightarrow\ A, ht\ \in\ (h\ \in\ P) \rightarrow\ Q\ (t\,h), he\ \in\ (h\ \in\ \neg\,P) \rightarrow\ Q\,(e\,h)) \in Q\,(ifD\,A\,P\,c\,t\,e) \end{array}$$

$undistIfD\; A\; Q\; P\; (\mathbf{inl}\; isP)\; t\; e\; ht\; he \equiv ht\; isP$

$undistIfD\; A\; Q\; P\; (\mathbf{inr}\; notP)\; t\; e\; ht\; he \equiv he\; notP$

Finally, we introduce an abbreviation to express in a compact way that the equality on a setoid is a decidable relation in the preceding sense, appealing as already mentioned to a library definition.

$decEq\; (A \in Setoid) \in Prop$

$decEq\; A \equiv DecidableRel\; (|\, A\, |)\; (==\; A)$

Operations on lists

Equality on lists is defined in the Alfa library, assuming that the base type is a setoid (this is of course needed for comparing the elements of the lists). By equality we mean here structural equality: the same elements in the same places.

$eqList\; (A \in Setoid) \in Rel\; (List\; (|\, A\, |))$

$eqList\; A \equiv (ListSetoid.List\; A)\; .Equal$

List membership is also defined for lists over a setoid.

$member\; (A \in Setoid, a \in |\, A\, |, L \in List\; (|\, A\, |)) \in Prop$

$member\; A\; a\; [\,] \equiv \perp$

$member\; A\; a\; (x : xs) \equiv ==\; A\; a\; x \vee member\; A\; a\; xs$

Next we define a predicate over lists of elements of a setoid, expressing the property that the list doesn't contain any repeated elements:

$noDupls\; (A \in Setoid, L \in List\; (|\, A\, |)) \in Prop$

$noDupls\; A\; [\,] \equiv T$

$noDupls\; A\; (x : xs) \equiv \neg\; (member\; A\; x\; xs)\; \&\; noDupls\; A\; xs$

The operation of adding an element to a list over a setoid is defined in such a way that the element is not added if it is found while traversing the list. When it is added, this is done at the end of the list, after making sure it wasn't in the list already:

$addElem\; (A \in Setoid, dEq \in decEq\; A, a \in |\, A\, |, L \in List\; (|\, A\, |)) \in List\; (|\, A\, |)$

$$addElem\,A\,dEq\,a\,[\,] \equiv a\ :\ [\,]$$

$$addElem\,A\,dEq\,a\ (x\ :\ xs) \equiv \begin{array}{l} ifD \\ \ (List\ (|\,A\,|)) \\ \ (==\ A\,a\,x) \\ \ (dEq\,a\,x) \\ \ \lambda\,h\ \rightarrow\ L \\ \ \lambda\,h\ \rightarrow\ x\ :\ addElem\,A\,dEq\,a\,xs \end{array}$$

We define an operation to remove an element from a list, with the behaviour that only the first copy of the element found in the list is removed. Since the intended use is that we apply this operation to lists without duplicates, it will in effect remove the element from the list completely.

$$rmvElem\ (A\ \in\ Setoid, dEq\ \in\ decEq\,A, a\ \in\ |\,A\,|, L\ \in\ List\ (|\,A\,|)) \in List\ (|\,A\,|)$$

$$rmvElem\,A\,dEq\,a\,[\,] \equiv [\,]$$

$$rmvElem\,A\,dEq\,a\ (x\ :\ xs) \equiv \begin{array}{l} ifD \\ \ (List\ (|\,A\,|)) \\ \ (==\ A\,a\,x) \\ \ (dEq\,a\,x) \\ \ \lambda\,h\ \rightarrow\ xs \\ \ \lambda\,h\ \rightarrow\ x\ :\ rmvElem\,A\,dEq\,a\,xs \end{array}$$

Next we define operations for adding and removing all the elements of a first list to/from a second list. They simply iterate the preceding two operations through the first argument list:

$$addElems\ (A\ \in\ Setoid, dEq\ \in\ decEq\,A, M, L\ \in\ List\ (|\,A\,|)) \in List\ (|\,A\,|)$$

$$addElems\,A\,dEq\,[\,]\,L \equiv L$$
$$addElems\,A\,dEq\ (x\ :\ xs)\ L \equiv addElem\,A\,dEq\,x\ (addElems\,A\,dEq\,xs\,L)$$

$$rmvElems\ (A\ \in\ Setoid, dEq\ \in\ decEq\,A, M, L\ \in\ List\ (|\,A\,|)) \in List\ (|\,A\,|)$$

$$rmvElems\,A\,dEq\,[\,]\,L \equiv L$$
$$rmvElems\,A\,dEq\ (x\ :\ xs)\ L \equiv rmvElem\,A\,dEq\,x\ (rmvElems\,A\,dEq\,xs\,L)$$

A test for the sublist property is now defined, with the meaning that l is a sublist of l' iff all its elements are members of l'. This is done keeping in mind that we would wish to have a subset relation available for using collections as a data type.

$$subList\ (A\ \in\ Setoid, L1, L2\ \in\ List\ (|\,A\,|)) \in Prop$$

$subList\ A\ []\ L2 \equiv T$

$subList\ A\ (x\ :\ xs)\ L2 \equiv member\ A\ x\ L2\ \&\ subList\ A\ xs\ L2$

Given the previous definition, the corresponding one for verifying that two lists have the same elements (not counting repetitions, that is once more as if they were sets) follows:

$sameList\ (A\ \in\ Setoid, L1, L2\ \in\ List\ (|\ A\ |)) \in Prop$

$sameList\ A\ L1\ L2 \equiv subList\ A\ L1\ L2\ \&\ subList\ A\ L2\ L1$

The next definition provides a way to establish that all the elements of a list satisfy a certain predicate, that is to say, a "for all the elements" quantifier. A similar "exists an element" operation could be defined following the same shape.

$allList\ (A\ \in\ Set, P\ \in\ Pred\ A, L\ \in\ List\ A) \in Prop$

$allList\ A\ P\ []\equiv T$

$allList\ A\ P\ (x\ :\ xs) \equiv P\ x\ \&\ allList\ A\ P\ xs$

Next is an operation to map a list along a function, but that differs from the usual list map combinator in that the result list is not allowed to have repetitions. That is to say, this will be the underlying function implementing the idea of map for collections. For achieving this, the mapped elements are added through *addElem* instead of consing. While the abstract finite set interface presented before lacks a map operation (which could be defined by means of the abstract fold operation), we still present this operation here since it was an interesting case of formalising collection concepts in type theory during our general work.

$mapList\ (A\ \in\ Set, B\ \in\ Setoid, dEq\ \in\ decEq\ B, f\ \in\ A\ \rightarrow\ |\ B\ |, L\ \in\ List\ A) \in List\ (|\ B\ |)$

$mapList\ A\ B\ dEq\ f\ [] \equiv []$

$mapList\ A\ B\ dEq\ f\ (x\ :\ xs) \equiv addElem\ B\ dEq\ (f\ x)\ (mapList\ A\ B\ dEq\ f\ xs)$

A filter operation on lists is also provided, with the assumption that the predicate the elements are tested against is a decidable one (in the same sense as decidability is used through all our formalisation of collections here):

$filterList\ (A\ \in\ Set, P\ \in\ Pred\ A, decP\ \in\ DecidablePred\ A\ P, L\ \in\ List\ A) \in List\ A$

$filterList\ A\ P\ decP\ [] \equiv []$

$$\mathit{filterList}\ A\ P\ \mathit{decP}\ (x : \mathit{xs}) \equiv \begin{array}{l} \mathit{ifD} \\ \quad (\mathit{List}\ A) \\ \quad (P\ x) \\ \quad (\mathit{decP}\ x) \\ \quad \lambda\ \mathit{isP}\ \rightarrow\ x : \mathit{filterList}\ A\ P\ \mathit{decP}\ \mathit{xs} \\ \quad \lambda\ \mathit{notP}\ \rightarrow\ \mathit{filterList}\ A\ P\ \mathit{decP}\ \mathit{xs} \end{array}$$

List intersection is defined by filtering out from the first list those elements that are not present in the second list (for the use of *decMember*, please see the immediately following subsection):

$\mathit{intersLists}\ (A \in \mathit{Setoid}, \mathit{dEq} \in \mathit{decEq}\ A, L1, L2 \in \mathit{List}\ (|\,A\,|)) \in \mathit{List}\ (|\,A\,|)$

$$\mathit{intersLists}\ A\ \mathit{dEq}\ L1\ L2 \equiv \begin{array}{l} \mathit{filterList} \\ \quad (|\,A\,|) \\ \quad \lambda\, a\ \rightarrow\ \mathit{member}\ A\ a\ L2 \\ \quad \lambda\, a\ \rightarrow\ \mathit{decMember}\ A\ \mathit{dEq}\ a\ L2 \\ \quad L1 \end{array}$$

Auxiliary properties

We need to use some auxiliary lemmas relating lists and decidability. First we show that list membership allows replacing equals for equals in its arguments. This will be handy for reasoning about properties of list operations.

$\mathit{substMember} \left(\begin{array}{l} A \in \mathit{Setoid}, a1, a2 \in |\,A\,|, \mathit{eqA} \in == A\ a1\ a2, L1, L2 \in \mathit{List} \\ (|\,A\,|), \mathit{eqL} \in \mathit{eqList}\ A\ L1\ L2, h \in \mathit{member}\ A\ a1\ L1 \end{array} \right)$
$\in \mathit{member}\ A\ a2\ L2$

$\mathit{substMember}\ A\ a1\ a2\ \mathit{eqA}\ [\,]\ L2\ \mathit{eqL}\ h \equiv \mathit{elimAbsurd}\ (\mathit{member}\ A\ a2\ L2)\ h$

$\mathit{substMember}\ A\ a1\ a2\ \mathit{eqA}\ (x : \mathit{xs})\ [\,]\ \mathit{eqL}\ h \equiv \mathit{eqL}$

$\mathit{substMember}\ A\ a1\ a2\ \mathit{eqA}\ (x : \mathit{xs})\ (x' : \mathit{xs}')\ \mathit{eqL}\ (\mathbf{inl}\ \mathit{eq}) \equiv$
$\quad \mathbf{inl}\ (A.\mathit{tran}\ a2\ x\ x'\ (A.\mathit{tran}\ a2\ a1\ x\ (A.\mathit{sym}\ a1\ a2\ \mathit{eqA})\ \mathit{eq})\ \mathit{eqL}.\mathit{fst})$

$\mathit{substMember}\ A\ a1\ a2\ \mathit{eqA}\ (x : \mathit{xs})\ (x' : \mathit{xs}')\ \mathit{eqL}\ (\mathbf{inr}\ \mathit{neq}) \equiv$
$\quad \mathbf{inr}\ (\mathit{substMember}\ A\ a1\ a2\ \mathit{eqA}\ \mathit{xs}\ \mathit{xs}'\ \mathit{eqL}.\mathit{snd}\ \mathit{neq})$

The next property shows that when we have lists over the elements of a setoid, and the setoid's equality is decidable, then list membership is also a decidable operation.

$\mathit{decMember}\ (A \in \mathit{Setoid}, \mathit{dEq} \in \mathit{decEq}\ A, a \in |\,A\,|, L \in \mathit{List}\ (|\,A\,|)) \in$
$\mathit{Decidable}\ (\mathit{member}\ A\ a\ L)$

```
decMember A dEq a [] ≡ inr λ h → h

decMember A dEq a (x : xs) ≡
 ifD
  (Decidable (member A a (x : xs)))
  (== A a x)
  (dEq a x)
  λ eq → inl (inl eq)
            ifD
             (Decidable (member A a (x : xs)))
             (member A a xs)
             (decMember A dEq a xs)
             λ inTl → inl (inr inTl)
  λ neq →                             elimOr
                                       (== A a x)
                                       (member A a xs)
             λ notInTl → inr λ h →     ⊥
                                       λ h′ → neq h′
                                       λ h′ → notInTl h′
                                       h
```

Finally, three abbreviations are used to provide reflexivity, symmetry and transitivity properties for the *eqList* equality on lists, by picking the appropriate pieces of the Alfa library.

$isReflEqList\ (A \in Setoid) \in Reflexive\ (List\ (|A|))\ (eqList\ A)$

$isReflEqList\ A \equiv (ListSetoid.List\ A)\ .ref$

$isSymEqList\ (A \in Setoid) \in Symmetrical\ (List\ (|A|))\ (eqList\ A)$

$isSymEqList\ A \equiv (ListSetoid.List\ A)\ .sym$

$isTransEqList\ (A \in Setoid) \in Transitive\ (List\ (|A|))\ (eqList\ A)$

$isTransEqList\ A \equiv (ListSetoid.List\ A)\ .tran$

Proofs that operations produce collections

The proofs needed are of the sort "show that this operation applied to a list without duplicates produces a list without duplicates". Once we build all the required proofs, we can put together the definitions for collections by pairing the operations with their corresponding proofs. As a remark to the reader, when we say "adding an element", we mean by using the special operation for that purpose defined previously.

We begin by showing that if something is not in a list, adding a different element to that list doesn't make it suddenly appear, of course:

$$lemNonMembAdd\left(\begin{array}{l} A \in Setoid, dEq \in decEq\,A, a, x \in |A| \\ xs \in List\ (|A|), h \in \neg\ (member\,A\,x\,xs), neq \in \neg\ (==\ A\,a\,x) \end{array}\right) \in$$
$$\neg\ (member\,A\,x\ (addElem\,A\,dEq\,a\,xs))$$

$$lemNonMembAdd\,A\,dEq\,a\,x\,[\,]\,h\,neq \equiv \lambda\,h' \rightarrow \begin{array}{l} \textbf{case}\ h'\ \textbf{of} \\ \quad \textbf{inl}\,inHd \rightarrow neq\ (A.sym\,x\,a\,inHd) \\ \quad \textbf{inr}\,inTl \rightarrow inTl \end{array}$$

$$\begin{array}{l}
lemNonMembAdd\,A\,dEq\,a\,x\ (x' : xs')\ h\,neq \equiv \\
\quad undistIfD \\
\qquad (List\ (|A|)) \\
\qquad \lambda\,L \rightarrow \neg\ (member\,A\,x\,L) \\
\qquad (==\ A\,a\,x') \\
\qquad (dEq\,a\,x') \\
\qquad \lambda\,h' \rightarrow x' : xs' \\
\qquad \lambda\,h' \rightarrow x' : addElem\,A\,dEq\,a\,xs' \\
\qquad \lambda\,h' \rightarrow h \\
\qquad\qquad\qquad elimOr \\
\qquad\qquad\qquad\ (==\ A\,x\,x') \\
\qquad\qquad\qquad\ (member\,A\,x\ (addElem\,A\,dEq\,a\,xs')) \\
\qquad \lambda\,h'\,h0 \rightarrow \quad \bot \\
\qquad\qquad\qquad\ \lambda\,h1 \rightarrow h\ (\textbf{inl}\,h1) \\
\qquad\qquad\qquad\ (lemNonMembAdd\,A\,dEq\,a\,x\,xs'\,\lambda\,h1 \rightarrow h\ (\textbf{inr}\,h1)\ neq) \\
\qquad\qquad\qquad\ h0
\end{array}$$

Adding an element to a list without duplications doesn't produce duplications. This is because *addElem* only adds an element if it is not already there.

$$\begin{array}{l} thNoDupAddElem \\ (A \in Setoid, dEq \in decEq\,A, a \in |A|, L \in List\ (|A|), ndL \in noDupls\,A\,L) \in \\ noDupls\,A\ (addElem\,A\,dEq\,a\,L) \end{array}$$

$$thNoDupAddElem\,A\,dEq\,a\,[\,]\,ndL \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} fst \equiv \lambda\,h \rightarrow h \\ snd \equiv ndL \end{array}\right. \end{array}$$

$$thNoDupAddElem\,A\,dEq\,a\ (x : xs)\ ndL \equiv$$

$$
\begin{array}{l}
\mathit{undistIfD} \\
\quad (\mathit{List}\ (|\, A \,|)) \\
\quad \lambda\, L \rightarrow \mathit{noDupls}\, A\, L \\
\quad (==\ A\, a\, x) \\
\quad (\mathit{dEq}\, a\, x) \\
\quad \lambda\, \mathit{eq} \rightarrow x : \mathit{xs} \\
\quad \lambda\, \mathit{neq} \rightarrow x : \mathit{addElem}\, A\, \mathit{dEq}\, a\, \mathit{xs} \\
\quad \lambda\, \mathit{eq} \rightarrow \mathit{ndL} \\
\quad \lambda\, \mathit{neq} \rightarrow \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} \mathit{fst} \equiv \mathit{lemNonMembAdd}\, A\, \mathit{dEq}\, a\, x\, \mathit{xs}\, \mathit{ndL.fst}\, \mathit{neq} \\ \mathit{snd} \equiv \mathit{thNoDupAddElem}\, A\, \mathit{dEq}\, a\, \mathit{xs}\, \mathit{ndL.snd} \end{array} \right. \end{array}
\end{array}
$$

The same holds for adding a whole list of elements to a list without duplications: we don't get any duplications in the resulting list. Once more, this follows from the way *addElem* is defined.

$$
\begin{array}{l}
\mathit{thNoDupAddElems} \\
(A \in \mathit{Setoid}, \mathit{dEq} \in \mathit{decEq}\, A, L1, L2 \in \mathit{List}\ (|\, A \,|)\,, \mathit{nd} \in \mathit{noDupls}\, A\, L2) \in \\
\mathit{noDupls}\, A\ \ (\mathit{addElems}\, A\, \mathit{dEq}\, L1\, L2)
\end{array}
$$

$$\mathit{thNoDupAddElems}\, A\, \mathit{dEq}\, [\,]\, L2\, \mathit{nd} \equiv \mathit{nd}$$

$$
\mathit{thNoDupAddElems}\, A\, \mathit{dEq}\ (x : \mathit{xs})\ L2\, \mathit{nd} \equiv \begin{array}{l}
\mathit{thNoDupAddElem} \\
\quad A \\
\quad \mathit{dEq} \\
\quad x \\
\quad (\mathit{addElems}\, A\, \mathit{dEq}\, \mathit{xs}\, L2) \\
\quad (\mathit{thNoDupAddElems}\, A\, \mathit{dEq}\, \mathit{xs}\, L2\, \mathit{nd})
\end{array}
$$

If something is not an element of a list, then it will still not be an element after removing some arbitrary element from that list:

$$
\begin{array}{l}
\mathit{lemNonMembRmv} \\
(A \in \mathit{Setoid}, \mathit{dEq} \in \mathit{decEq}\, A, a, x \in |\, A \,|, \mathit{xs} \in \mathit{List}\ (|\, A \,|)\,, h \in \neg\ (\mathit{member}\, A\, x\, \mathit{xs})) \in \\
\neg\ (\mathit{member}\, A\, x\ (\mathit{rmvElem}\, A\, \mathit{dEq}\, a\, \mathit{xs}))
\end{array}
$$

$$\mathit{lemNonMembRmv}\, A\, \mathit{dEq}\, a\, x\, [\,]\, h \equiv h$$

$$\mathit{lemNonMembRmv}\, A\, \mathit{dEq}\, a\, x\ (x' : \mathit{xs}')\ h\ \equiv$$

$$
\begin{array}{l}
\mathit{undistIfD} \\
\quad (\mathit{List}\ (|\,A\,|)) \\
\quad \lambda\, L \rightarrow \neg\ (\mathit{member}\ A\ x\ L) \\
\quad (==\ A\ a\ x') \\
\quad (\mathit{dEq}\ a\ x') \\
\quad \lambda\, \mathit{eq}' \rightarrow \mathit{xs}' \\
\quad \lambda\, \mathit{neq}' \rightarrow x' : \mathit{rmvElem}\ A\ \mathit{dEq}\ a\ \mathit{xs}' \\
\quad \lambda\, \mathit{eq}'\ h' \rightarrow h\ (\mathbf{inr}\ h') \\
\qquad\qquad\quad \mathit{elimOr} \\
\qquad\qquad\qquad (==\ A\ x\ x') \\
\qquad\qquad\qquad (\mathit{member}\ A\ x\ (\mathit{rmvElem}\ A\ \mathit{dEq}\ a\ \mathit{xs}')) \\
\quad \lambda\, \mathit{neq}'\ h' \rightarrow \quad \bot \\
\qquad\qquad\qquad \lambda\, h0 \rightarrow h\ (\mathbf{inl}\ h0) \\
\qquad\qquad\qquad (\mathit{lemNonMembRmv}\ A\ \mathit{dEq}\ a\ x\ \mathit{xs}'\ \lambda\, h0 \rightarrow h\ (\mathbf{inr}\ h0)) \\
\qquad\qquad\qquad h'
\end{array}
$$

The operation of removing an element from a list without duplicates doesn't produce a result with duplicates:

$$
\begin{array}{l}
\mathit{thNoDupRmvElem} \\
(A \in \mathit{Setoid}, \mathit{dEq} \in \mathit{decEq}\ A, a \in |\,A\,|, L \in \mathit{List}\ (|\,A\,|)\,, \mathit{ndL} \in \mathit{noDupls}\ A\ L) \in \\
\mathit{noDupls}\ A\ \ (\mathit{rmvElem}\ A\ \mathit{dEq}\ a\ L)
\end{array}
$$

$$\mathit{thNoDupRmvElem}\ A\ \mathit{dEq}\ a\ []\ \mathit{ndL} \equiv \mathbf{tt}$$

$$
\begin{array}{l}
\mathit{thNoDupRmvElem}\ A\ \mathit{dEq}\ a\ (x : \mathit{xs})\ \mathit{ndL} \equiv \\
\quad \mathit{undistIfD} \\
\qquad (\mathit{List}\ (|\,A\,|)) \\
\qquad \lambda\, L' \rightarrow \mathit{noDupls}\ A\ L' \\
\qquad (==\ A\ a\ x) \\
\qquad (\mathit{dEq}\ a\ x) \\
\qquad \lambda\, \mathit{eq} \rightarrow \mathit{xs} \\
\qquad \lambda\, \mathit{neq} \rightarrow x : \mathit{rmvElem}\ A\ \mathit{dEq}\ a\ \mathit{xs} \\
\qquad \lambda\, \mathit{eq} \rightarrow \mathit{ndL.snd} \\
\qquad \lambda\, \mathit{neq} \rightarrow \quad \begin{array}{l} \mathbf{struct} \\ \left[\begin{array}{l} \mathit{fst} \equiv \mathit{lemNonMembRmv}\ A\ \mathit{dEq}\ a\ x\ \mathit{xs}\ \mathit{ndL.fst} \\ \mathit{snd} \equiv \mathit{thNoDupRmvElem}\ A\ \mathit{dEq}\ a\ \mathit{xs}\ \mathit{ndL.snd} \end{array}\right. \end{array}
\end{array}
$$

And a similar result holds for removing a whole list of elements from a list without duplicates:

$$
\begin{array}{l}
\mathit{thNoDupRmvElems} \\
(A \in \mathit{Setoid}, \mathit{dEq} \in \mathit{decEq}\ A, L1, L2 \in \mathit{List}\ (|\,A\,|)\,, \mathit{nd} \in \mathit{noDupls}\ A\ L2) \in \\
\mathit{noDupls}\ A\ \ (\mathit{rmvElems}\ A\ \mathit{dEq}\ L1\ L2)
\end{array}
$$

$thNoDupRmvElems\,A\,dEq\,[\,]\,L2\,nd \equiv nd$

$$thNoDupRmvElems\,A\,dEq\,(x : xs)\,L2\,nd \equiv \begin{array}{l} thNoDupRmvElem \\ \;A \\ \;dEq \\ \;x \\ \;(rmvElems\,A\,dEq\,xs\,L2) \\ \;(thNoDupRmvElems\,A\,dEq\,xs\,L2\,nd) \end{array}$$

This auxiliary lemma says that if a list is a sublist of another, it will still be a sublist of the result of consing any element to that other list.

$lemSubListTl\,(A \in Setoid, x \in |A|, L1, L2 \in List\,(|A|), h \in subList\,A\,L1\,L2) \in subList\,A\,L1\,(x : L2)$

$lemSubListTl\,A\,x\,[\,]\,L2\,h \equiv \mathbf{tt}$

$$lemSubListTl\,A\,x\,(x' : xs)\,L2\,h \equiv \begin{array}{l} \mathbf{struct} \\ \left[\begin{array}{l} fst \equiv \mathbf{inr}\,h.fst \\ snd \equiv lemSubListTl\,A\,x\,xs\,L2\,h.snd \end{array}\right. \end{array}$$

We also need the property that "being a sublist of" is a reflexive relation on lists:

$isReflSubList\,(A \in Setoid) \in Reflexive\,(List\,(|A|))\,(subList\,A)$

$$\begin{array}{l} isReflSubList\,A \equiv \lambda L \;\rightarrow \\ \quad \mathbf{case}\;L\;\mathbf{of} \\ \qquad [\,] \;\rightarrow\; \mathbf{tt} \\ \qquad x : xs \;\rightarrow\; \begin{array}{l} \mathbf{struct} \\ \left[\begin{array}{l} fst \equiv \mathbf{inl}\,(A.ref\,x) \\ snd \equiv lemSubListTl\,A\,x\,xs\,xs\,(isReflSubList\,A\,xs) \end{array}\right. \end{array} \end{array}$$

Another useful fact is that membership is in a way transitive over the sublist relation, or more exactly, it is preseved by the sublist relation. The heading of the definition shows the exact sense in which this happens:

$lemTransMemb$
$(A \in Setoid, x \in |A|, L1, L2 \in List\,(|A|), hM \in member\,A\,x\,L1, hS \in subList\,A\,L1\,L2)$
$\in member\,A\,x\,L2$

$lemTransMemb\,A\,x\,[\,]\,L2\,hM\,hS \equiv elimAbsurd\,(member\,A\,x\,L2)\,hM$

$lemTransMemb\,A\,x\,(x' : xs)\,L2\,(\mathbf{inl}\,inHd)\,hS \equiv$
$\quad substMember\,A\,x'\,x\,(A.sym\,x\,x'\,inHd)\,L2\,L2\,(isReflEqList\,A\,L2)\,hS.fst$

$lemTransMemb\,A\,x\,(x' : xs)\,L2\,(\mathbf{inr}\,inTl)\,hS \equiv lemTransMemb\,A\,x\,xs\,L2\,inTl\,hS.snd$

Also, the sublist relation is transitive:

$isTransSubList\ (A \in Setoid) \in Transitive\ (List\ (|A|))\ (subList\ A)$

$$
\begin{array}{l}
isTransSubList\ A \equiv \lambda\ L1\ L2\ L3\ h12\ h23 \rightarrow \\
\quad \textbf{case}\ L1\ \textbf{of} \\
\qquad [\,] \rightarrow \textbf{tt} \\
\qquad x : xs \rightarrow \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv lemTransMemb\ A\ x\ L2\ L3\ h12.fst\ h23 \\ snd \equiv isTransSubList\ A\ xs\ L2\ L3\ h12.snd\ h23 \end{array}\right.\end{array}
\end{array}
$$

The "same list" relation (that is, list considered the same when they contain the same elements, regardless of order or number of repetitions), is a reflexive, symmetric and transitive relation:

$isReflSameList\ (A \in Setoid) \in Reflexive\ (List\ (|A|))\ (sameList\ A)$

$$
isReflSameList\ A \equiv \lambda\ L \rightarrow \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv isReflSubList\ A\ L \\ snd \equiv fst \end{array}\right.\end{array}
$$

$isSymmSameList\ (A \in Setoid) \in Symmetrical\ (List\ (|A|))\ (sameList\ A)$

$$
isSymmSameList\ A \equiv \lambda\ L1\ L2\ h \rightarrow \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} fst \equiv h.snd \\ snd \equiv h.fst \end{array}\right.\end{array}
$$

$isTransSameList\ (A \in Setoid) \in Transitive\ (List\ (|A|))\ (sameList\ A)$

$$
\begin{array}{l}
isTransSameList\ A \equiv \lambda\ L1\ L2\ L3\ h12\ h23 \rightarrow \\
\quad \textbf{struct} \\
\qquad \left[\begin{array}{l} fst \equiv isTransSubList\ A\ L1\ L2\ L3\ h12.fst\ h23.fst \\ snd \equiv isTransSubList\ A\ L3\ L2\ L1\ h23.snd\ h12.snd \end{array}\right.
\end{array}
$$

The special map operation produces a list without duplicates:

$thNoDupsMapList\ (A \in Set, B \in Setoid, dEq \in decEq\ B, f \in A \rightarrow |B|, L \in List\ A) \in noDupls\ B\ (mapList\ A\ B\ dEq\ f\ L)$

$thNoDupsMapList\ A\ B\ dEq\ f\ [\,] \equiv \textbf{tt}$

$$
thNoDupsMapList\ A\ B\ dEq\ f\ (x : xs) \equiv \begin{array}{l} thNoDupAddElem \\ \quad B \\ \quad dEq \\ \quad (f\,x) \\ \quad (mapList\ A\ B\ dEq\ f\ xs) \\ \quad (thNoDupsMapList\ A\ B\ dEq\ f\ xs) \end{array}
$$

If we add an element to a list, then that element will be a member of the resulting list:

$$
\begin{array}{l}
\mathit{thMembAdd}\ (A \in \mathit{Setoid}, \mathit{dEq} \in \mathit{decEq}\, A, a \in |\, A\,|, L \in \mathit{List}\ (|\, A\,|)) \in \\
\mathit{member}\, A\, a\ (\mathit{addElem}\, A\, \mathit{dEq}\, a\, L)
\end{array}
$$

$$
\mathit{thMembAdd}\, A\, \mathit{dEq}\, a\, [\,] \equiv \mathbf{inl}\ (A.\mathit{ref}\, a)
$$

$$
\begin{array}{ll}
 & \mathit{undistIfD} \\
 & \quad (\mathit{List}\ (|\, A\,|)) \\
 & \quad \lambda\, L \ \rightarrow\ \mathit{member}\, A\, a\, L \\
 & \quad (==\ A\, a\, x) \\
\mathit{thMembAdd}\, A\, \mathit{dEq}\, a\ (x : xs) \equiv & \quad (\mathit{dEq}\, a\, x) \\
 & \quad \lambda\, \mathit{eq} \ \rightarrow\ L \\
 & \quad \lambda\, \mathit{neq} \ \rightarrow\ x : \mathit{addElem}\, A\, \mathit{dEq}\, a\, xs \\
 & \quad \lambda\, \mathit{eq} \ \rightarrow\ \mathbf{inl}\, \mathit{eq} \\
 & \quad \lambda\, \mathit{neq} \ \rightarrow\ \mathbf{inr}\ (\mathit{thMembAdd}\, A\, \mathit{dEq}\, a\, xs)
\end{array}
$$

And conversely, if we remove an element from a list, it will not be a member of the resulting list:

$$
\begin{array}{l}
\mathit{thNotMembRmv} \\
(A \in \mathit{Setoid}, \mathit{dEq} \in \mathit{decEq}\, A, a \in |\, A\,|, L \in \mathit{List}\ (|\, A\,|)\,, \mathit{ndL} \in \mathit{noDupls}\, A\, L) \in \\
\neg\ (\mathit{member}\, A\, a\ (\mathit{rmvElem}\, A\, \mathit{dEq}\, a\, L))
\end{array}
$$

$$
\mathit{thNotMembRmv}\, A\, \mathit{dEq}\, a\, [\,]\, \mathit{ndL} \equiv \lambda\, h \ \rightarrow\ h
$$

$$
\begin{array}{l}
\mathit{thNotMembRmv}\, A\, \mathit{dEq}\, a\ (x : xs)\ \mathit{ndL}\ \equiv \\
\qquad \mathit{undistIfD} \\
\qquad\quad (\mathit{List}\ (|\, A\,|)) \\
\qquad\quad \lambda\, L \ \rightarrow\ \neg\ (\mathit{member}\, A\, a\, L) \\
\qquad\quad (==\ A\, a\, x) \\
\qquad\quad (\mathit{dEq}\, a\, x) \\
\qquad\quad \lambda\, \mathit{eq} \ \rightarrow\ xs \\
\qquad\quad \lambda\, \mathit{neq} \ \rightarrow\ x : \mathit{rmvElem}\, A\, \mathit{dEq}\, a\, xs \\
\qquad\quad \lambda\, \mathit{eq}\, h \ \rightarrow\ \mathit{ndL}.\mathit{fst}\ (\mathit{substMember}\, A\, a\, x\, \mathit{eq}\, xs\, xs\ (\mathit{isReflEqList}\, A\, xs)\ h) \\
\qquad\qquad\qquad\quad \mathit{elimOr} \\
\qquad\qquad\qquad\quad\ (==\ A\, a\, x) \\
\qquad\qquad\qquad\quad\ (\mathit{member}\, A\, a\ (\mathit{rmvElem}\, A\, \mathit{dEq}\, a\, xs)) \\
\qquad\quad \lambda\, \mathit{neq}\, h \ \rightarrow\ \quad \bot \\
\qquad\qquad\qquad\quad\ \mathit{neq} \\
\qquad\qquad\qquad\quad\ (\mathit{thNotMembRmv}\, A\, \mathit{dEq}\, a\, xs\, \mathit{ndL}.\mathit{snd}) \\
\qquad\qquad\qquad\quad\ h
\end{array}
$$

If an element is not a member of a list, it will not be a member of applying a filter operation to that list:

$$\mathit{lemNonMembFilter}\left(\begin{array}{l} A \in \mathit{Setoid}, P \in \mathit{Pred}\ (|A|), \mathit{decP} \in \mathit{DecidablePred}\ (|A|)\ P, \\ x \in |A|, \mathit{xs} \in \mathit{List}\ (|A|), h \in \neg\ (\mathit{member}\ A\ x\ \mathit{xs}) \end{array}\right) \in$$
$$\neg\ (\mathit{member}\ A\ x\ (\mathit{filterList}\ (|A|)\ P\ \mathit{decP}\ \mathit{xs}))$$

$$\begin{array}{l}
\mathit{lemNonMembFilter}\ A\ P\ \mathit{decP}\ x\ []\ h \equiv h \\[4pt]
\mathit{lemNonMembFilter}\ A\ P\ \mathit{decP}\ x\ (x' : \mathit{xs}')\ h \equiv \\
\quad \mathit{undistIfD} \\
\qquad (\mathit{List}\ (|A|)) \\
\qquad \lambda L \to \neg\ (\mathit{member}\ A\ x\ L) \\
\qquad (P\ x') \\
\qquad (\mathit{decP}\ x') \\
\qquad \lambda\ \mathit{isP} \to x' : \mathit{filterList}\ (|A|)\ P\ \mathit{decP}\ \mathit{xs}' \\
\qquad \lambda\ \mathit{notP} \to \mathit{filterList}\ (|A|)\ P\ \mathit{decP}\ \mathit{xs}' \\
\qquad \lambda\ \mathit{isP}\ h' \to \\
\qquad\quad \mathit{elimOr} \\
\qquad\quad\ (==\ A\ x\ x') \\
\qquad\quad\ (\mathit{member}\ A\ x\ (\mathit{filterList}\ (|A|)\ P\ \mathit{decP}\ \mathit{xs}')) \\
\qquad\quad\ \bot \\
\qquad\quad\ \lambda\ \mathit{inHd} \to h\ (\mathbf{inl}\ \mathit{inHd}) \\
\qquad\quad\ \lambda\ \mathit{inTl} \to \mathit{lemNonMembFilter}\ A\ P\ \mathit{decP}\ x\ \mathit{xs}'\ \lambda\ h0 \to h\ (\mathbf{inr}\ h0)\ \mathit{inTl} \\
\qquad\quad\ h' \\
\qquad \lambda\ \mathit{notP} \to \mathit{lemNonMembFilter}\ A\ P\ \mathit{decP}\ x\ \mathit{xs}'\ \lambda\ h' \to h\ (\mathbf{inr}\ h')
\end{array}$$

The filter operation produces a list without duplicates:

$$\mathit{thNoDupFilter}\left(\begin{array}{l} A \in \mathit{Setoid}, P \in \mathit{Pred}\ (|A|), \mathit{decP} \in \mathit{DecidablePred}\ (|A|)\ P \\ L \in \mathit{List}\ (|A|), \mathit{ndL} \in \mathit{noDupls}\ A\ L \end{array}\right) \in$$
$$\mathit{noDupls}\ A\ (\mathit{filterList}\ (|A|)\ P\ \mathit{decP}\ L)$$

$$\begin{array}{l}
\mathit{thNoDupFilter}\ A\ P\ \mathit{decP}\ []\ \mathit{ndL} \equiv \mathbf{tt} \\[4pt]
\mathit{thNoDupFilter}\ A\ P\ \mathit{decP}\ (x : \mathit{xs})\ \mathit{ndL} \equiv \\
\quad \mathit{undistIfD} \\
\qquad (\mathit{List}\ (|A|)) \\
\qquad \lambda L' \to \mathit{noDupls}\ A\ L' \\
\qquad (P\ x) \\
\qquad (\mathit{decP}\ x) \\
\qquad \lambda\ \mathit{isP} \to x : \mathit{filterList}\ (|A|)\ P\ \mathit{decP}\ \mathit{xs} \\
\qquad \lambda\ \mathit{notP} \to \mathit{filterList}\ (|A|)\ P\ \mathit{decP}\ \mathit{xs} \\
\qquad \lambda\ \mathit{isP} \to \quad \mathbf{struct}\left[\begin{array}{l} \mathit{fst} \equiv \mathit{lemNonMembFilter}\ A\ P\ \mathit{decP}\ x\ \mathit{xs}\ \mathit{ndL}.\mathit{fst} \\ \mathit{snd} \equiv \mathit{thNoDupFilter}\ A\ P\ \mathit{decP}\ \mathit{xs}\ \mathit{ndL}.\mathit{snd} \end{array}\right. \\
\qquad \lambda\ \mathit{notP} \to \mathit{thNoDupFilter}\ A\ P\ \mathit{decP}\ \mathit{xs}\ \mathit{ndL}.\mathit{snd}
\end{array}$$

The list intersection operation also produces a list without duplicates:

$thNoDupInters\ (A \in Setoid, dEq \in decEq\ A, L1, L2 \in List\ (|\ A\ |), nd \in noDupls\ A\ L1)$
$\in noDupls\ A\ (intersLists\ A\ dEq\ L1\ L2)$

$$thNoDupInters\ A\ dEq\ L1\ L2\ nd \equiv \begin{array}{l} thNoDupFilter \\ \ A \\ \ \lambda\, a \rightarrow member\ A\ a\ L2 \\ \ \lambda\, a \rightarrow decMember\ A\ dEq\ a\ L2 \\ \ L1 \\ \ nd \end{array}$$

The sublist relation is decidable:

$decSubList\ (A \in Setoid, dEq \in decEq\ A, L1, L2 \in List\ (|\ A\ |)) \in$
$Decidable\ (subList\ A\ L1\ L2)$

$decSubList\ A\ dEq\ []\ L2 \equiv \mathbf{inl\ tt}$

$$decSubList\ A\ dEq\ (x : xs)\ L2 \equiv \begin{array}{l} decAnd \\ \ (member\ A\ x\ L2) \\ \ (subList\ A\ xs\ L2) \\ \ (decMember\ A\ dEq\ x\ L2) \\ \ (decSubList\ A\ dEq\ xs\ L2) \end{array}$$

And the “same list” relation is also decidable:

$decSameList\ (A \in Setoid, dEq \in decEq\ A, L1, L2 \in List\ (|\ A\ |)) \in$
$Decidable\ (sameList\ A\ L1\ L2)$

$$decSameList\ A\ dEq\ L1\ L2 \equiv \begin{array}{l} decAnd \\ \ (subList\ A\ L1\ L2) \\ \ (subList\ A\ L2\ L1) \\ \ (decSubList\ A\ dEq\ L1\ L2) \\ \ (decSubList\ A\ dEq\ L2\ L1) \end{array}$$

Finally, it is decidable if the “for all elements” operation holds or not, for any list:

$decAllList\ (A \in Set, P \in Pred\ A, decP \in DecidablePred\ A\ P, L \in List\ A) \in$
$Decidable\ (allList\ A\ P\ L)$

$decAllList\ A\ P\ decP\ []\ \equiv \mathbf{inl\ tt}$

$$
\mathit{decAllList}\;A\;P\;\mathit{decP}\;(x\;:\;xs) \equiv
\begin{array}{l}
\mathit{ifD'} \\
\quad (\mathit{Decidable}\;(\mathit{allList}\;A\;P\;(x\;:\;xs))) \\
\quad (P\;x) \\
\quad (\mathit{decP}\;x) \\
\quad \lambda\,\mathit{isP} \;\rightarrow\;
\begin{array}{l}
\mathit{ifD'} \\
\quad (\mathit{Decidable}\;(\mathit{allList}\;A\;P\;(x\;:\;xs))) \\
\quad (\mathit{allList}\;A\;P\;xs) \\
\quad (\mathit{decAllList}\;A\;P\;\mathit{decP}\;xs) \\
\quad \lambda\,\mathit{isTl} \;\rightarrow\; \mathbf{inl}\left(\begin{array}{l} \mathbf{struct} \\ \left[\begin{array}{l} \mathit{fst} \equiv \mathit{isP} \\ \mathit{snd} \equiv \mathit{isTl} \end{array}\right.\end{array}\right) \\
\quad \lambda\,\mathit{notTl} \;\rightarrow\; \mathbf{inr}\,\lambda\,h \;\rightarrow\; \mathit{notTl}\;h.\mathit{snd}
\end{array} \\
\quad \lambda\,\mathit{notP} \;\rightarrow\; \mathbf{inr}\,\lambda\,h \;\rightarrow\; \mathit{notP}\;h.\mathit{fst}
\end{array}
$$

6.7.4 Putting together the collection operations

We can now provide collection operations of the right types, by bundling together the list operations and their properties showed before.

$\mathit{inColl}\;(A \in \mathit{Setoid}, a \in |A|, C \in \mathit{Coll}\;A) \in \mathit{Prop}$

$\mathit{inColl}\;A\;a\;C \equiv \mathit{member}\;A\;a\;C.\mathit{elems}$

$\mathit{addElemColl}\;(A \in \mathit{Setoid}, \mathit{dEq} \in \mathit{decEq}\;A, a \in |A|, C \in \mathit{Coll}\;A) \in \mathit{Coll}\;A$

$$
\mathit{addElemColl}\;A\;\mathit{dEq}\;a\;C \equiv
\begin{array}{l}
\mathbf{struct} \\
\left[\begin{array}{l}
\mathit{elems} \equiv \mathit{addElem}\;A\;\mathit{dEq}\;a\;C.\mathit{elems} \\
\mathit{ppty} \equiv \mathit{thNoDupAddElem}\;A\;\mathit{dEq}\;a\;C.\mathit{elems}\;C.\mathit{ppty}
\end{array}\right.
\end{array}
$$

$\mathit{addElemsColl}\;(A \in \mathit{Setoid}, \mathit{dEq} \in \mathit{decEq}\;A, L \in \mathit{List}\;(|A|), C \in \mathit{Coll}\;A) \in \mathit{Coll}\;A$

$$
\mathit{addElemsColl}\;A\;\mathit{dEq}\;L\;C \equiv
\begin{array}{l}
\mathbf{struct} \\
\left[\begin{array}{l}
\mathit{elems} \equiv \mathit{addElems}\;A\;\mathit{dEq}\;L\;C.\mathit{elems} \\
\mathit{ppty} \equiv \mathit{thNoDupAddElems}\;A\;\mathit{dEq}\;L\;C.\mathit{elems}\;C.\mathit{ppty}
\end{array}\right.
\end{array}
$$

$\mathit{rmvElemColl}\;(A \in \mathit{Setoid}, \mathit{dEq} \in \mathit{decEq}\;A, a \in |A|, C \in \mathit{Coll}\;A) \in \mathit{Coll}\;A$

$$
\mathit{rmvElemColl}\;A\;\mathit{dEq}\;a\;C \equiv
\begin{array}{l}
\mathbf{struct} \\
\left[\begin{array}{l}
\mathit{elems} \equiv \mathit{rmvElem}\;A\;\mathit{dEq}\;a\;C.\mathit{elems} \\
\mathit{ppty} \equiv \mathit{thNoDupRmvElem}\;A\;\mathit{dEq}\;a\;C.\mathit{elems}\;C.\mathit{ppty}
\end{array}\right.
\end{array}
$$

$\mathit{rmvElemsColl}\;(A \in \mathit{Setoid}, \mathit{dEq} \in \mathit{decEq}\;A, L \in \mathit{List}\;(|A|), C \in \mathit{Coll}\;A) \in \mathit{Coll}\;A$

$$rmvElemsColl\ A\ dEq\ L\ C \equiv \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} elems \equiv rmvElems\ A\ dEq\ L\ C.elems \\ ppty \equiv thNoDupRmvElems\ A\ dEq\ L\ C.elems\ C.ppty \end{array}\right.\end{array}$$

$unionColl\ (A \in Setoid, dEq \in decEq\ A, C1, C2 \in Coll\ A) \in Coll\ A$

$unionColl\ A\ dEq\ C1\ C2 \equiv addElemsColl\ A\ dEq\ C1.elems\ C2$

$diffColl\ (A \in Setoid, dEq \in decEq\ A, C1, C2 \in Coll\ A) \in Coll\ A$

$diffColl\ A\ dEq\ C1\ C2 \equiv rmvElemsColl\ A\ dEq\ C2.elems\ C1$

$filterColl$
$(A \in Setoid, dEq \in decEq\ A, P \in Pred\ (|A|), decP \in DecidablePred\ (|A|)\ P, C \in Coll\ A)$
$\in Coll\ A$

$$filterColl\ A\ dEq\ P\ decP\ C \equiv \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} elems \equiv filterList\ (|A|)\ P\ decP\ C.elems \\ ppty \equiv thNoDupFilter\ A\ P\ decP\ C.elems\ C.ppty \end{array}\right.\end{array}$$

$intersColl\ (A \in Setoid, dEq \in decEq\ A, C1, C2 \in Coll\ A) \in Coll\ A$

$$intersColl\ A\ dEq\ C1\ C2 \equiv \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} elems \equiv intersLists\ A\ dEq\ C1.elems\ C2.elems \\ ppty \equiv thNoDupInters\ A\ dEq\ C1.elems\ C2.elems\ C1.ppty \end{array}\right.\end{array}$$

For the operations corresponding to subset inclusion and set equality, we also provide convenient abbreviations for their properties. This actually establishes the collections over some base type form a setoid, which present the possibility of constructing collections of collections (something that is not possible for abstract finite sets):

$subColl\ (A \in Setoid, C1, C2 \in Coll\ A) \in Prop$

$subColl\ A\ C1\ C2 \equiv subList\ A\ C1.elems\ C2.elems$

$isReflSubColl\ (A \in Setoid) \in Reflexive\ (Coll\ A)\ (subColl\ A)$

$isReflSubColl\ A \equiv \lambda\, C \rightarrow isReflSubList\ A\ C.elems$

$isTransSubColl\ (A \in Setoid) \in Transitive\ (Coll\ A)\ (subColl\ A)$

$isTransSubColl\ A \equiv \lambda\, C1\ C2\ C3 \rightarrow isTransSubList\ A\ C1.elems\ C2.elems\ C3.elems$

$eqColl\ (A \ \in\ Setoid, C1, C2\ \in\ Coll\ A) \in Prop$

$eqColl\ A\ C1\ C2 \equiv sameList\ A\ C1.elems\ C2.elems$

$isReflEqColl\ (A\ \in\ Setoid) \in Reflexive\ (Coll\ A)\ (eqColl\ A)$

$isReflEqColl\ A \equiv \lambda\ C\ \rightarrow\ isReflSameList\ A\ C.elems$

$isSymmEqColl\ (A\ \in\ Setoid) \in Symmetrical\ (Coll\ A)\ (eqColl\ A)$

$isSymmEqColl\ A \equiv \lambda\ C1\ C2\ \rightarrow\ isSymmSameList\ A\ C1.elems\ C2.elems$

$isTransEqColl\ (A\ \in\ Setoid) \in Transitive\ (Coll\ A)\ (eqColl\ A)$

$isTransEqColl\ A \equiv \lambda\ C1\ C2\ C3\ \rightarrow\ isTransSameList\ A\ C1.elems\ C2.elems\ C3.elems$

$CollSetoid\ (A\ \in\ Setoid) \in Setoid$

$$CollSetoid\ A \equiv \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l} Elem \equiv Coll\ A \\ Equal \equiv eqColl\ A \\ ref \equiv isReflEqColl\ A \\ sym \equiv isSymmEqColl\ A \\ tran \equiv isTransEqColl\ A \end{array}\right.\end{array}$$

$decInColl\ (A\ \in\ Setoid, dEq\ \in\ decEq\ A, a\ \in\ |\,A\,|, C\ \in\ Coll\ A) \in Decidable\ (inColl\ A\ a\ C)$

$decInColl\ A\ dEq\ a\ C \equiv decMember\ A\ dEq\ a\ C.elems$

$decSubColl\ (A\ \in\ Setoid, dEq\ \in\ decEq\ A, C1, C2\ \in\ Coll\ A) \in$
$Decidable\ (subColl\ A\ C1\ C2)$

$decSubColl\ A\ dEq\ C1\ C2 \equiv decSubList\ A\ dEq\ C1.elems\ C2.elems$

$decEqColl\ (A\ \in\ Setoid, dEq\ \in\ decEq\ A, C1, C2\ \in\ Coll\ A) \in Decidable\ (eqColl\ A\ C1\ C2)$

$decEqColl\ A\ dEq\ C1\ C2 \equiv decSameList\ A\ dEq\ C1.elems\ C2.elems$

$allInColl\ (A\ \in\ Setoid, P\ \in\ Pred\ (|\,A\,|), C\ \in\ Coll\ A) \in Prop$

$allInColl\ A\ P\ C \equiv allList\ (|\,A\,|)\ P\ C.elems$

$$decAllInColl\ (A \in Setoid, P \in Pred\ (|A|), decP \in DecidablePred\ (|A|)\ P, C \in Coll\ A) \\ \in Decidable\ (allInColl\ A\ P\ C)$$

$$decAllInColl\ A\ P\ decP\ C \equiv decAllList\ (|A|)\ P\ decP\ C.elems$$

$$mapColl\ (A, B \in Setoid, dEq \in decEq\ B, f \in (|A|) \to |B|, C \in Coll\ A) \in Coll\ B$$

$$mapColl\ A\ B\ dEq\ f\ C \equiv \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} elems \equiv mapList\ (|A|)\ B\ dEq\ f\ C.elems \\ ppty \equiv thNoDupsMapList\ (|A|)\ B\ dEq\ f\ ?_0 \end{array} \right. \end{array}$$

6.7.5 Collections are a finite set implementation

We will now show that collections can implement quite straightforwardly the operations required by the abstract finite set signature. We don't deal here with the axioms that are also part of the abstract specification.

First we set up a few auxiliary properties to be used in the definitions of the operations for the implementation. We start by showing that the equality for the setoid that underlies every ordered type is a decidable equality. This fact is needed to convert from the propositional-style decidability to the Boolean-valued one in some of the instantiations of the abstract signature for finite sets.

$$isDecEqSFO\ (A \in OrderedType) \in decEq\ (SetoidFromOrder\ A)$$

$$\begin{array}{l} isDecEqSFO\ A \equiv \lambda\, x1\ x2\ \to \\ \quad \textbf{case}\ A.compare\ x1\ x2\ \textbf{of} \\ \qquad \textbf{Lt}\ h\ \to\ \textbf{inr}\ (A.ltNotEq\ x1\ x2\ h) \\ \qquad \textbf{Eq}\ h\ \to\ \textbf{inl}\ h \\ \qquad \textbf{Gt}\ h\ \to\ \textbf{inr}\ \lambda\, h'\ \to\ A.ltNotEq\ x2\ x1\ h\ (isSymEqOrdType\ A\ x1\ x2\ h') \end{array}$$

The next two auxiliary definitions are provided so that we can define the minimum operation, which wasn't part of the collection operations as presented before.

$$min\ (A \in OrderedType, x, y \in A.t) \in A.t$$

$$min\ A\ x\ y \equiv \begin{array}{l} \textbf{case}\ A.compare\ x\ y\ \textbf{of} \\ \quad \textbf{Lt}\ h\ \to\ x \\ \quad \textbf{Eq}\ h\ \to\ x \\ \quad \textbf{Gt}\ h\ \to\ y \end{array}$$

$$minElem\ (A \in OrderedType, l \in List\ A.t) \in Maybe\ A.t$$

$$minElem\ A\ [\,] \equiv \textbf{Fail}$$

$$minElem\ A\ (x : xs) \equiv \textbf{case}\ minElem\ A\ xs\ \textbf{of} \left\{ \begin{array}{l} \textbf{Ok}\ a\ \to\ \textbf{Ok}\ (min\ A\ x\ a) \\ \textbf{Fail}\ \to\ \textbf{Ok}\ x \end{array} \right\}$$

Now we can present the actual instantiations *CollAbsFinSets* of the fields for the abstract signature *ASet* for finite sets.

$CollAbsFinSets\ (A \in OrderedType) \in ASet\ A$

Abstract sets are represented as collections. The conversion from ordered type to setoid will be present throughout all the field definitions, since collections are not defined over an ordered type. Considering the definition of *ASet A*, to show that *CollAbsFinSets* is an instance of *ASet* we need to instantiate all the fields in *ASet*. The first field is the underlying set, the "carrier" of the abstract finite set.

$$\left[\begin{array}{l} t \in Set \\ \quad t \equiv Coll\ (SetoidFromOrder\ A) \end{array}\right.$$

The second field is the empty abstract finite set. It is defined as the empty collection.

$$\left[\begin{array}{l} empty \in t \\ \quad empty \equiv \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} elems \equiv [] \\ ppty \equiv \textbf{tt} \end{array}\right.\end{array} \end{array}\right.$$

Then we define the test for empty abstract finite sets. Notice the use of *dec2bool* for transforming a propositional decidable test into a Boolean one. This will be done throughout the several field definitions that follow, too.

$$\left[\begin{array}{l} isEmpty \in t \rightarrow SET.Bool \\ \quad isEmpty \equiv \lambda s \rightarrow \begin{array}{l} dec2bool \\ \quad (eqColl\ (SetoidFromOrder\ A)\ s\ empty) \\ \quad (decEqColl\ (SetoidFromOrder\ A)\ (isDecEqSFO\ A)\ s\ empty) \end{array} \end{array}\right.$$

Membership also requires converting the type of test into a Boolean valued one:

$$\left[\begin{array}{l} mem \in A.t \rightarrow t \rightarrow SET.Bool \\ \quad mem \equiv \lambda x\, s \rightarrow \begin{array}{l} dec2bool \\ \quad (inColl\ (SetoidFromOrder\ A)\ x\, s) \\ \quad (decInColl\ (SetoidFromOrder\ A)\ (isDecEqSFO\ A)\ x\, s) \end{array} \end{array}\right.$$

Adding an element is implemented straighforwardly too:

$$\left[\begin{array}{l} add \in A.t \rightarrow t \rightarrow t \\ \quad add \equiv addElemColl\ (SetoidFromOrder\ A)\ (isDecEqSFO\ A) \end{array}\right.$$

Singleton sets are formed by constructing the collection with that element as the only one present in the underlying list:

$$\left[\begin{array}{l} singleton \in A.t \rightarrow t \\ \quad singleton \equiv \lambda x \rightarrow \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} elems \equiv x : [] \\ ppty \equiv \textbf{struct} \left\{ \left[\begin{array}{l} fst \equiv \lambda h \rightarrow h \\ snd \equiv \textbf{tt} \end{array}\right. \right\} \end{array}\right.\end{array} \end{array}\right.$$

Now follow more operations that are implemented straightforwardly in terms of the collection operations:

$$\left[\begin{array}{l} remove \in A.t \to t \to t \\ \quad remove \equiv rmvElemColl\ (SetoidFromOrder\ A)\ (isDecEqSFO\ A) \end{array}\right.$$

$$\left[\begin{array}{l} union \in t \to t \to t \\ \quad union \equiv unionColl\ (SetoidFromOrder\ A)\ (isDecEqSFO\ A) \end{array}\right.$$

$$\left[\begin{array}{l} inter \in t \to t \to t \\ \quad inter \equiv intersColl\ (SetoidFromOrder\ A)\ (isDecEqSFO\ A) \end{array}\right.$$

$$\left[\begin{array}{l} diff \in t \to t \to t \\ \quad diff \equiv diffColl\ (SetoidFromOrder\ A)\ (isDecEqSFO\ A) \end{array}\right.$$

$$\left[\begin{array}{l} eq \in t \to t \to Prop \\ \quad eq \equiv eqColl\ (SetoidFromOrder\ A) \end{array}\right.$$

$$\left[\begin{array}{l} lt \in t \to t \to Prop \\ \quad lt \equiv subColl\ (SetoidFromOrder\ A) \end{array}\right.$$

Fold is implemented by just using the list folding combinator provided by the Alfa library. Its type is as follows:

$$\left[\ fold \in (A' \in Set) \to (A.t \to A' \to A') \to t \to A' \to A'\right.$$

The universal quantifier test for finite sets comes from the one for collections, with the adequate conversion of the tests into Boolean valued ones:

$$\left[\begin{array}{l} forAll \in (A.t \to Bool) \to t \to Bool \\ \quad forAll \equiv \lambda\, P\, s \to \begin{array}{l} dec2bool \\ \quad (allInColl\ (SetoidFromOrder\ A)\ \lambda\, x \to |P\,x|\, s) \\ \quad \left(\begin{array}{l} decAllInColl \\ \quad (SetoidFromOrder\ A) \\ \quad \lambda\, x \to |P\,x| \\ \quad \lambda\, x \to decTrue\ (P\,x) \\ \quad s \end{array}\right) \end{array} \end{array}\right.$$

The existential one can be defined in pretty much the same style and shape, and we omit it here, just as the corresponding collection operation was also omitted in the previous discussion. The filter operation for finite sets follows, and it should be noted that in this case we apply a conversion from Boolean tests to propositional valued decidable ones, since that was the kind used in the collection operation:

$$\left[\begin{array}{l} filter \in (A.t \to Bool) \to t \to t \\ \quad filter \equiv \lambda\, P\, s \to \begin{array}{l} filterColl \\ \quad (SetoidFromOrder\ A) \\ \quad (isDecEqSFO\ A) \\ \quad \lambda\, x \to |P\,x| \\ \quad \lambda\, x \to decTrue\ (P\,x) \\ \quad s \end{array} \end{array}\right.$$

The partition operation can be defined in terms of two filter operations:

$$\left[\begin{array}{l} \mathit{partition} \in (A.t \to \mathit{Bool}) \to t \to t \times t \\ \mathit{partition} \equiv \lambda\, P\, s \to \begin{array}{l}\textbf{struct} \\ \left[\begin{array}{l} \mathit{fst} \equiv \mathit{filter}\, P\, s \\ \mathit{snd} \equiv \mathit{filter}\, \lambda\, x \to \neg\, (P\, x)\, s \end{array}\right.\end{array} \end{array}\right.$$

Cardinality of a finite set can be obtained from the length operation for lists provided by the library. Notice that this gives the correct count since collections are by definition built without duplicates.

$$\left[\begin{array}{l} \mathit{cardinal} \in t \to \mathit{Nat} \\ \mathit{cardinal} \equiv \lambda\, s \to \mathit{length}\, A.t\, s.\mathit{elems} \end{array}\right.$$

The "listification" of a finite set's elements is trivial in this representation, of course. We just access the underlying list.

$$\left[\begin{array}{l} \mathit{elements} \in t \to \mathit{List}\, A.t \\ \mathit{elements} \equiv \lambda\, s \to s.\mathit{elems} \end{array}\right.$$

The definition of minimum follows, using the auxiliary operations defined at the beginning of this section:

$$\left[\begin{array}{l} \mathit{minElt} \in t \to \mathit{Maybe}\, A.t \\ \mathit{minElt} \equiv \lambda\, s \to \mathit{minElem}\, A\, s.\mathit{elems} \end{array}\right.$$

A very similar definition could be used for maximum. Finally we define the choice operation. Since by the nature of Alfa and collections this must be deterministic, a simple solution is just to grab the head of the underlying list if it is nonempty, and to fail otherwise:

$$\left[\begin{array}{l} \mathit{choose} \in t \to \mathit{Maybe}\, A.t \\ \mathit{choose} \equiv \lambda\, s \to \begin{array}{l} \mathit{elimList}\, A.t\ (\lambda\, l \to \mathit{Maybe}\, A.t) \\ \quad \textbf{Fail} \\ \quad (\lambda\, x\, \mathit{xs}\, h \to \textbf{Ok}\, x\, s.\mathit{elems}) \end{array} \end{array}\right.$$

6.7.6 Defining databases in terms of abstract finite sets

We are now ready to define our notion of databases and their operations purely in terms of the abstract finite set interface. For our purposes, we represent schemes as finite sets *Fin n*. Thus we have the following definitions:

$\mathit{Scheme} \in \mathit{Set}$

$\mathit{Scheme} \equiv \mathit{Nat}$

$\mathit{Typing}\ (n \in \mathit{Scheme}) \in \mathit{Type}$

$\mathit{Typing}\, n \equiv \mathit{Fin}\, n \to \mathit{OrderedType}$

$$\mathit{Tuple}\begin{pmatrix} n \in \mathit{Scheme} \\ A \in \mathit{Typing}\, n \end{pmatrix} \in \mathit{Set}$$

$$\mathit{Tuple}\, n\, A \equiv (i \in \mathit{Fin}\, n) \to (A\, i)\,.t$$

Since we later need to break up the tuples into the head and the rest, and give the results meaningful types, we also introduce the following auxiliary functions:

$$\mathit{restTyping}\begin{pmatrix} n \in \mathit{Scheme} \\ A \in \mathit{Typing}\ (\mathbf{suc}\, n) \end{pmatrix} \in \mathit{Typing}\, n$$

$$\mathit{restTyping}\, n\, A \equiv \lambda\, i \to A\ (\mathbf{suc}\, i)$$

$$\mathit{restTuple}\begin{pmatrix} n \in \mathit{Scheme} \\ A \in \mathit{Typing}\ (\mathbf{suc}\, n) \\ x \in \mathit{Tuple}\ (\mathbf{suc}\, n)\ A \end{pmatrix} \in \mathit{Tuple}\, n\ (\mathit{restTyping}\, n\, A)$$

$$\mathit{restTuple}\, n\, A\, x \equiv \lambda\, i \to x\ (\mathbf{suc}\, i)$$

The lexicographically ordered type of tuples We need to prove that tuples form an ordered type and so can be the basis on which an abstract finite set can be defined, thus we must provide both an equality and a less-than relation for tuples. Since the definition of ordered types keeps these relations tightly matched with the compare operation, we must be careful. The shape of these two definitions is such that the definition of compare can proceed from them.

$$\mathit{eqTuple}\begin{pmatrix} n \in \mathit{Scheme} \\ A \in \mathit{Typing}\, n \\ x \in \mathit{Tuple}\, n\, A \\ y \in \mathit{Tuple}\, n\, A \end{pmatrix} \in \mathit{Prop}$$

$$\mathit{eqTuple}\ \mathbf{zer}\, A\, x\, y \equiv T$$

$$\begin{aligned} &\mathit{eqTuple}\ (\mathbf{suc}\, m)\ A\, x\, y \equiv \\ &\quad (A\, \mathbf{zer})\,.\mathit{eq}\ (x\, \mathbf{zer})\ (y\, \mathbf{zer})\ \& \\ &\qquad \mathit{eqTuple}\, m\ (\mathit{restTyping}\, m\, A)\ (\mathit{restTuple}\, m\, A\, x)\ (\mathit{restTuple}\, m\, A\, y) \end{aligned}$$

$$\mathit{ltTuple}\begin{pmatrix} n \in \mathit{Scheme} \\ A \in \mathit{Typing}\, n \\ x \in \mathit{Tuple}\, n\, A \\ y \in \mathit{Tuple}\, n\, A \end{pmatrix} \in \mathit{Prop}$$

$$\mathit{ltTuple}\ \mathbf{zer}\, A\, x\, y \equiv \bot$$

$$\begin{aligned} &\mathit{ltTuple}\ (\mathbf{suc}\, m)\ A\, x\, y \equiv \\ &\quad (A\, \mathbf{zer})\,.\mathit{lt}\ (x\, \mathbf{zer})\ (y\, \mathbf{zer}) \vee \\ &\qquad ((A\, \mathbf{zer})\,.\mathit{eq}\ (x\, \mathbf{zer})\ (y\, \mathbf{zer}) \\ &\qquad\quad \&\ \mathit{ltTuple}\, m\ (\mathit{restTyping}\, m\, A)\ (\mathit{restTuple}\, m\, A\, x)\ (\mathit{restTuple}\, m\, A\, y)) \end{aligned}$$

It is important to remark that the less-than relation we use is the lexicographic ordering: that is, viewing the tuples as strings of elements of the corresponding domains. This is a useful ordering, used in practice and also good for our purposes since it gives a total ordering of the tuples, as required by the definition of ordered type. We can now define the compare operation by recursion over the tuples viewed as strings in this way.

$$compTuple\begin{pmatrix} n \in Scheme \\ A \in Typing\,n \\ x \in Tuple\,n\,A \\ y \in Tuple\,n\,A \end{pmatrix} \in Compare\ (Tuple\,n\,A)\ (ltTuple\,n\,A)\ (eqTuple\,n\,A)\ x\,y$$

$$compTuple\,\mathbf{zer}\,A\,x\,y \equiv \mathbf{Eq}\,\mathbf{tt}$$

$$compTuple\ (\mathbf{suc}\,m)\ A\,x\,y \equiv$$

$$\begin{array}{l}
elimCompare \\
\quad (A\,\mathbf{zer})\,.t \\
\quad (A\,\mathbf{zer})\,.lt \\
\quad (A\,\mathbf{zer})\,.eq \\
\quad (x\,\mathbf{zer}) \\
\quad (y\,\mathbf{zer}) \\
\quad \lambda h\ \rightarrow\ Compare\ (Tuple\,n\,A)\ (ltTuple\ (\mathbf{suc}\,m)\ A)\ (eqTuple\ (\mathbf{suc}\,m)\ A)\ x\,y \\
\quad \lambda h\ \rightarrow\ \mathbf{Lt}\ (\mathbf{inl}\,h) \\
\quad \lambda h\ \rightarrow\ \begin{array}{l}
elimCompare \\
\quad (i\ \in\ Fin\,m) \rightarrow\ (restTyping\,m\,A\,i)\,.t \\
\quad (ltTuple\,m\ (restTyping\,m\,A)) \\
\quad (eqTuple\,m\ (restTyping\,m\,A)) \\
\quad (restTuple\,m\,A\,x) \\
\quad (restTuple\,m\,A\,y) \\
\quad \lambda h'\ \rightarrow\ Compare\ (Tuple\,n\,A)\ (ltTuple\,n\,A)\ (eqTuple\,n\,A)\ x\,y \\
\quad \lambda h'\ \rightarrow\ \mathbf{Lt}\left(\mathbf{inr}\left(\begin{array}{l}\mathbf{struct} \\ \quad \left[\begin{array}{l} fst \equiv h \\ snd \equiv h' \end{array}\right.\end{array}\right)\right) \\
\quad \lambda h'\ \rightarrow\ \mathbf{Eq}\left(\begin{array}{l}\mathbf{struct} \\ \quad \left[\begin{array}{l} fst \equiv h \\ snd \equiv h' \end{array}\right.\end{array}\right) \\
\quad \lambda h'\ \rightarrow\ \mathbf{Gt}\left(\mathbf{inr}\left(\begin{array}{l}\mathbf{struct} \\ \quad \left[\begin{array}{l} fst \equiv (A\,\mathbf{zer})\,.eqSym\ (x\,\mathbf{zer})\ (y\,\mathbf{zer})\ h \\ snd \equiv h' \end{array}\right.\end{array}\right)\right) \\
\quad (compTuple\,m\ (restTyping\,m\,A)\ (restTuple\,m\,A\,x)\ (restTuple\,m\,A\,y))
\end{array} \\
\quad \lambda h\ \rightarrow\ \mathbf{Gt}\ (\mathbf{inl}\,h) \\
\quad ((A\,\mathbf{zer})\,.compare\ (x\,\mathbf{zer})\ (y\,\mathbf{zer}))
\end{array}$$

Now we show that equality of tuples is reflexive, symmetric and transitive, as required by the definition of an ordered type:

$$\mathit{isRefEqTuple}\left(\begin{array}{l} n \in \mathit{Scheme} \\ A \in \mathit{Typing}\, n \end{array}\right) \in \mathit{Reflexive}\ (\mathit{Tuple}\, n\, A)\ (\mathit{eqTuple}\, n\, A)$$

$$\begin{array}{l}
\mathit{isRefEqTuple}\, n\, A \equiv \lambda\, x\ \rightarrow \\
\quad \textbf{case}\ n\ \textbf{of} \\
\qquad \textbf{zer}\ \rightarrow\ \textbf{tt} \\
\qquad \textbf{suc}\, m\ \rightarrow\ \textbf{struct}\left[\begin{array}{l} \mathit{fst} \equiv (A\ \textbf{zer})\, .\mathit{eqRefl}\ (x\ \textbf{zer}) \\ \mathit{snd} \equiv \mathit{isRefEqTuple}\, m\ (\mathit{restTyping}\, m\, A)\ (\mathit{restTuple}\, m\, A\, x) \end{array}\right.
\end{array}$$

$$\mathit{isSymEqTuple}\left(\begin{array}{l} n \in \mathit{Scheme} \\ A \in \mathit{Typing}\, n \end{array}\right) \in \mathit{Symmetrical}\ (\mathit{Tuple}\, n\, A)\ (\mathit{eqTuple}\, n\, A)$$

$$\begin{array}{l}
\mathit{isSymEqTuple}\, n\, A \equiv \lambda\, x\, y\, h\ \rightarrow \\
\quad \textbf{case}\ n\ \textbf{of} \\
\qquad \textbf{zer}\ \rightarrow\ \textbf{tt} \\
\qquad \textbf{suc}\, m\ \rightarrow\ \textbf{struct}\left[\begin{array}{l} \mathit{fst} \equiv (A\ \textbf{zer})\, .\mathit{eqSym}\ (x\ \textbf{zer})\ (y\ \textbf{zer})\ h.\mathit{fst} \\ \mathit{snd} \equiv \mathit{isSymEqTuple}\, m\ (\mathit{restTyping}\, m\, A) \\ \qquad\quad (\mathit{restTuple}\, m\, A\, x)\ (\mathit{restTuple}\, m\, A\, y)\ h.\mathit{snd} \end{array}\right.
\end{array}$$

$$\mathit{isTranEqTuple}\left(\begin{array}{l} n \in \mathit{Scheme} \\ A \in \mathit{Typing}\, n \end{array}\right) \in \mathit{Transitive}\ (\mathit{Tuple}\, n\, A)\ (\mathit{eqTuple}\, n\, A)$$

$$\begin{array}{l}
\mathit{isTranEqTuple}\, n\, A \equiv \lambda\, x\, y\, z\, h\, h'\ \rightarrow \\
\quad \textbf{case}\ n\ \textbf{of} \\
\qquad \textbf{zer}\ \rightarrow\ \textbf{tt} \\
\qquad \textbf{suc}\, m\ \rightarrow\ \textbf{struct}\left[\begin{array}{l} \mathit{fst} \equiv (A\ \textbf{zer})\, .\mathit{eqTran}\ (x\ \textbf{zer})\ (y\ \textbf{zer})\ (z\ \textbf{zer})\ h.\mathit{fst}\ h'.\mathit{fst} \\ \mathit{snd} \equiv \mathit{isTranEqTuple}\, m\ (\mathit{restTyping}\, m\, A)\ (\mathit{restTuple}\, m\, A\, x) \\ \qquad\qquad (\mathit{restTuple}\, m\, A\, y)\ (\mathit{restTuple}\, m\, A\, z)\ h.\mathit{snd}\ h'.\mathit{snd} \end{array}\right.
\end{array}$$

Now we must show that the less-than lexicographic ordering on tuples is transitive, another requirement for an ordered type:

$$\mathit{isTranLtTuple}\left(\begin{array}{l} n \in \mathit{Scheme} \\ A \in \mathit{Typing}\, n \end{array}\right) \in \mathit{Transitive}\ (\mathit{Tuple}\, n\, A)\ (\mathit{ltTuple}\, n\, A)$$

```
isTranLtTuple n A ≡ λ x y z h h′ →
 case n of
   zer → h
   suc m →
       case h of
         inl hLhd →
             case h′ of
               inl hLhd′ → inl ((A zer).ltTran (x zer) (y zer) (z zer) hLhd hLhd′)
               inr hEhd′ → inl ((A zer).ltSubst (x zer) (x zer) (y zer) (z zer)
                                 ((A zer).eqRefl (x zer)) hEhd′.fst hLhd)
         inr hEhd →
             case h′ of
               inl hLhd′ → inl ((A zer).ltSubst (y zer) (x zer) (z zer) (z zer)
                                 ((A zer).eqSym (x zer) (y zer) hEhd.fst)
                                 ((A zer).eqRefl (z zer)) hLhd′)
               inr hEhd′ → inr
                   struct
                    [ fst ≡ (A zer).eqTran (x zer) (y zer) (z zer)
                            hEhd.fst hEhd′.fst
                      snd ≡ isTranLtTuple m (restTyping m A) (restTuple m A x)
                             (restTuple m A y) (restTuple m A z) hEhd.snd hEhd′.snd
```

The next required proof (for tuples to be an ordered type) is that less-than is different from equality:

$$ltIsNotEq \begin{pmatrix} n \in Scheme \\ A \in Typing\, n \end{pmatrix} \in$$
$$(x, y \in Tuple\, n\, A) \to ltTuple\, n\, A\, x\, y \to \neg\, (eqTuple\, n\, A\, x\, y)$$

```
ltIsNotEq n A ≡ λ x y h →
 case n of
   zer → λ h′ → h
   suc m → λ h′ →
                     case h of
                       inl hL → (A zer).ltNotEq (x zer) (y zer) hL h′.fst
                       inr hE →  ltIsNotEq
                                  m
                                  (restTyping m A)
                                  (restTuple m A x)
                                  (restTuple m A y)
                                  hE.snd
                                  h′.snd
```

Finally, we must show that the lexicographic less-than is substitutive with respect to tuple equality:

$$ltIsSubst \begin{pmatrix} n \in Scheme \\ A \in Typing\, n \end{pmatrix} \in (x, x', y, y' \in\ Tuple\, n\, A) \to\ eqTuple\, n\, A\, x\, x' \ \to$$
$$eqTuple\, n\, A\, y\, y' \ \to\ ltTuple\, n\, A\, x\, y \ \to\ ltTuple\, n\, A\, x'\, y'$$

```
ltIsSubst n A ≡ λ x x′ y y′ hX hY h →
 case n of
   zer → h
   suc m →
      case (A zer).compare (x′ zer) (y′ zer) of
        Lt h0 → inl h0
        Eq h0 → inr
           struct
             [ fst ≡ h0 ]
             [ snd ≡
                 case h of
                   inl hL →
                      elimAbsurd
                       (ltTuple m (restTyping m A)
                           (restTuple m A x′) (restTuple m A y′))
                       ( (A zer).ltNotEq
                          (x′ zer) (y′ zer)
                          ((A zer).ltSubst (x zer) (x′ zer) (y zer) (y′ zer)
                                 hX.fst hY.fst hL)
                          h0                                              )
                           ltIsSubst
                             m (restTyping m A)
                   inr hE →    (restTuple m A x) (restTuple m A x′)
                             (restTuple m A y) (restTuple m A y′)
                             hX.sndhY.sndhE.snd
             ]
        Gt h0 →
          case h of
                     elimAbsurd
                      (ltTuple (suc m) A x′ y′)
             inl hL →  ( lemLtNotGt
                         (A zer) (y′ zer) (x′ zer) h0
                         ((A zer).ltSubst (x zer) (x′ zer) (y zer) (y′ zer)
                                hX.fst hY.fst hL)                          )
             inr hE →
                elimAbsurd
                 (ltTuple (suc m) A x′ y′)
                 ( (A zer).ltNotEq
                   (y′ zer) (x′ zer) h0
                   ( (A zer).eqTran
                      (y′ zer) (y zer) (x′ zer)
                      ((A zer).eqSym (y zer) (y′ zer) hY.fst)
                      ( (A zer).eqTran
                         (y zer) (x zer) (x′ zer)
                         ((A zer).eqSym (x zer) (y zer) hE.fst)
                         hX.fst                                 ) ) )
```

We can finally pack all these proofs together and define the ordered types of tuples over a specific scheme as follows:

$$\mathit{TupleOrd}\left(\begin{array}{l} n \in \mathit{Scheme} \\ A \in \mathit{Typing}\; n \end{array}\right) \in \mathit{OrderedType}$$

$$\mathit{TupleOrd}\; n\; A \equiv \quad \begin{array}{l} \textbf{struct} \\ \left[\begin{array}{l} t \equiv \mathit{Tuple}\; n\; A \\ \mathit{eq} \equiv \mathit{eqTuple}\; n\; A \\ \mathit{lt} \equiv \mathit{ltTuple}\; n\; A \\ \mathit{eqRefl} \equiv \mathit{isRefEqTuple}\; n\; A \\ \mathit{eqSym} \equiv \mathit{isSymEqTuple}\; n\; A \\ \mathit{eqTran} \equiv \mathit{isTranEqTuple}\; n\; A \\ \mathit{ltTran} \equiv \mathit{isTranLtTuple}\; n\; A \\ \mathit{ltNotEq} \equiv \mathit{ltIsNotEq}\; n\; A \\ \mathit{compare} \equiv \mathit{compTuple}\; n\; A \\ \mathit{ltSubst} \equiv \mathit{ltIsSubst}\; n\; A \end{array}\right. \end{array}$$

Databases and their operations We now proceed to define the abstract type of database relations over some scheme as the abstract finite sets on tuples over that scheme:

$$\mathit{DBSet}\left(\begin{array}{l} n \in \mathit{Scheme} \\ A \in \mathit{Typing}\; n \end{array}\right) \in \mathit{Type}$$

$$\mathit{DBSet}\; n\; A \equiv \mathit{ASet}\; (\mathit{TupleOrd}\; n\; A)$$

We can define for ease of access the following abbreviation for relations:

$$\mathit{DBRel}\left(\begin{array}{l} n \in \mathit{Scheme} \\ A \in \mathit{Typing}\; n \\ W \in \mathit{DBSet}\; n\; A \end{array}\right) \in \mathit{Set}$$

$$\mathit{DBRel}\; n\; A\; W \equiv W.t$$

Most database operations are immediately available as abstract set operations, for instance for the union of two relations over the same scheme:

$$\mathit{unionDBRel}\left(\begin{array}{l} n \in \mathit{Scheme} \\ A \in \mathit{Typing}\; n \\ W \in \mathit{DBSet}\; n\; A \\ R \in \mathit{DBRel}\; n\; A\; W \\ S \in \mathit{DBRel}\; n\; A\; W \end{array}\right) \in \mathit{DBRel}\; n\; A\; W$$

$unionDBRel\ n\ A\ W\ R\ S \equiv W.union\ R\ S$

Likewise for intersection, difference, equality, inclusion of relations, which are taken directly from the abstract set operations. Filter provides the immediate definition for the selection operation on a relation. It is interesting to notice that the abstract set signature is actually powerful enough to allow us to express the extended operators of relational algebra, such as minimum and counting, though we don't deal with such operators in the present discussion.

The only operation that requires any actual effort to define is the join of relations. We remind the reader of what was used in a preceding section to deal with joins on schemes of the form *Fin n*: we saw composite schemes as expressed by things of the form *Fin (m+n)*, with corresponding operations to take the first and second parts of a tuple so composed, and a fusion operation that produced from a tuple over *Fin (m+n)* and a tuple over *Fin (n+p)* that agreed over the *n* part a result tuple over *Fin (m+n+p)*.

For this, we first need to define the map function between two abstract finite sets:

$$mapASet \begin{pmatrix} A \in OrderedType \\ B \in OrderedType \\ A' \in ASet\ A \\ B' \in ASet\ B \\ f \in A.t \rightarrow B.t \end{pmatrix} \in A'.t \rightarrow B'.t$$

$$mapASet\ A\ B\ A'\ B'\ f \equiv \lambda\, x \rightarrow A'.fold\ B'.t\ (\lambda\, a\, y \rightarrow B'.add\ (f\, a)\ y)\ x\ B'.empty$$

The definition of join of R and S would then consist of two nested maps: we map over R a function that in turn takes the current tuple of R under consideration and maps in turn over S the tuple fusion operation. For this use, we would need to provide a definition of tuple fusion that returns an abstract finite set, this set being either empty if the two tuples which we compare don't agree on their common subschemes, and a singleton with the fused result tuple if they do agree.

6.7.7 Other implementations

In practice, several quite different data structures exist for implementing an abstract data type behaving as a finite set. Our preceding long development of collections is very illustrative of the way lists and similar data structures can implement the abstract data type. Two approaches which are quite different from lists are trees and hash structures for implementing finite sets.

Starting from the simplest binary search trees, all the way to some of the technical refinements of B+ trees, a wide variety of tree-based implementations of the finite set data abstraction are possible. Some of these are relatively straightforward to develop in Alfa, and even a few cases are available to learn about such efforts (see [59]). The particular style and idioms of our abstract signature are taken, as already mentioned, from the work of Filliâtre and Letouzey [29], where besides an implementation comparable to our collections idea, a few balanced trees-based ones are provided in all detail. Browsing through the proof code provided by the authors should be enough to convince a reader that for such advanced data structures the proof effort is quite considerable.

Regarding the other broad family of data structures, the ones using hash functions, there are again a good number of variants from the simplest static hash tables all the way to complex multidimensional hashing structures. The hash-based ones seem quite different from tree-based implementations with respect to a formalization effort. While the actual algorithms and data representations involved in hash tables are pretty straightforward, the interest of this kind of data structure comes from the fact that the hash function used to choose the storage position of the elements have statistical properties that make them efficient. Formalizing the correctness of a hash structure should be simple enough. On the other hand, their statistical properties would need a formalisation of probabilistic and random number concepts, and how far from the kind of language provided by a proof assistant in general (and Alfa in particular) such discourse is, we can only guess at the completely new character of such a formal proof development compared to the subjects investigated in the present work.

6.7.8 On attempting full generality

In the preceding sections we have dealt with tuples as functions, and the corresponding schemes were either arbitrary types or types intended to convey that the attributes of the scheme were listed in a fixed order. One may wonder if perhaps the abstract set interface makes it possible to turn schemes into sets of attributes, as is the usual approach in database textbooks on relational algebra. Here we will briefly explore that possibility.

To be able to form abstract sets of attributes, that is schemes, we need to have a global universe of attributes with an ordering, and hence a corresponding global typing function associating domains of data (themselves also ordered types) to each attribute label:

$Attributes \in Type$

$$Attributes \equiv \begin{array}{l} \textbf{sig} \\ \quad attr \in OrderedType \\ \quad dom \in attr.t \rightarrow OrderedType \end{array}$$

Now a scheme can be an abstract finite set of attribute names:

$Schemes\ (W \in Attributes) \in Type$

$Schemes\ W \equiv ASet\ W.attr$

A possible definition for a tuple is as a function going from the scheme to the corresponding domain of the argument attribute. However, the fact that the attribute actually is an element of the scheme can only be ensured by having the function receive as argument a proof object verifying that membership (introducing again the problem that there may be different proof objects). A tuple would then be

$$Tuple \begin{pmatrix} W \in Attributes \\ Z \in Schemes\ W \\ Sch \in Z.t \end{pmatrix} \in Set$$

$$Tuple\ W\ Z\ Sch \equiv (x \in W.attr.t) \to (|\ Z.mem\ x\ Sch\ |) \to (W.dom\ x).t$$

This representation would work fine for the purpose of altering the structure of tuples either by taking a subtuple or by adding another tuple over a disjoint scheme (like in the tuple fusion we used for joins). The problem with this approach is when we define the compare operation, which must be decidable. The definition of tuple equality can be done through a universal quantification without any trouble, but the same is not possible for tuple less-than. As the scheme is now a set, we need to make explicit what order to follow, and the most natural one is again a lexicographic ordering following the attribute names on the scheme. Now, to perform the element-wise comparison of the entries in two tuples over the same scheme, we would need a way to recurse in that lexicographic order, and this is not possible with the abstract set signature we have (notice that fold would not suffice for this kind of traversal of the set, and we can't do a recursive definition by case analysis on the result of min, where the recursive call would be done over the set difference of the scheme and the minimal attribute, since this would be an incorrect definition for type theory).

We need to find a different approach than using functions for tuples. An alternative kind of definition, used by Date in his "Third Manifesto" [21] for relational database theory, views tuples as (dependently typed) association sets, that is sets with elements of the shape (A_i, x_i) for A_i in the tuple's scheme and x_i in the data domain corresponding to A_i. Clearly there is a requirement that only one pair must be present for each A_i in the scheme. Such pairs are called *fields*, and are a Σ type:

$field\ (W \in Attributes) \in Set$

$$field\ W \equiv \begin{array}{l} \textbf{sig} \\ \quad lab \in W.attr.t \\ \quad val \in (W.dom\ lab).t \end{array}$$

Since we wish to form tuples by having a set of fields (with some consistency properties with respect to the scheme, as already mentioned), we define equality of fields:

$$eqField\begin{pmatrix} W \in Attributes \\ f1 \in field\,W \\ f2 \in field\,W \end{pmatrix} \in Bool$$

$$eqField\ W\ f1\ f2 \equiv$$

$$\begin{array}{l}
\quad elimCompare \\
\quad\quad W.attr.t \\
\quad\quad W.attr.lt \\
\quad\quad W.attr.eq \\
\quad\quad f1.lab \\
\quad\quad f2.lab \\
\quad\quad \lambda\, c \to Bool \\
\quad\quad \lambda\, hL \to \mathbf{false} \\
\quad\quad \lambda\, hE \to \\
\quad\quad\quad\quad elimCompare \\
\quad\quad\quad\quad\quad (W.dom\, f1.lab)\,.t \\
\quad\quad\quad\quad\quad (W.dom\, f1.lab)\,.lt \\
\quad\quad\quad\quad\quad (W.dom\, f1.lab)\,.eq \\
\quad\quad\quad\quad\quad f1.val \\
\quad\quad\quad\quad\quad ((lemDepOrderIso\ W.attr\ W.dom\, f1.lab\, f2.lab\ hE)\,.bwd\, f2.val) \\
\quad\quad\quad\quad\quad \lambda\, c' \to Bool \\
\quad\quad\quad\quad\quad \lambda\, hL' \to \mathbf{false} \\
\quad\quad\quad\quad\quad \lambda\, hE' \to \mathbf{true} \\
\quad\quad\quad\quad\quad \lambda\, hG' \to \mathbf{false} \\
\quad\quad\quad\quad\quad \begin{pmatrix} (W.dom\, f1.lab)\,.compare \\ \quad f1.val \\ \quad ((lemDepOrderIso\ W.attr\ W.dom\, f1.lab\, f2.lab\ hE)\,.bwd\, f2.val) \end{pmatrix} \\
\quad\quad \lambda\, hG \to \mathbf{false} \\
\quad\quad (W.attr.compare\, f1.lab\, f2.lab)
\end{array}$$

Besides the complex structure of the definition, we remark that it is needed to use auxiliary definitions and lemmas relating to isomorphisms between ordered types. The reason is that since the attribute names are literally different, Alfa treats the associated data domains as different, even in the situation where the compare elimination has established that the names are equal. And as a result of the need for this isomorphism, the equality on an ordered type needs to be changed from an equivalence relation into a reflexive substitutive one. The actual definitions follow:

$$OrderIso\begin{pmatrix} X \in OrderedType \\ Y \in OrderedType \end{pmatrix} \in Set$$

$$OrderIso\ X\ Y \equiv$$

sig

$$\begin{array}{l}
\mathit{fwd} \in X.t \to Y.t \\
\mathit{bwd} \in Y.t \to X.t \\
\mathit{fwdSub} \in (x1, x2 \in X.t) \to X.eq\, x1\, x2 \to Y.eq\, (\mathit{fwd}\, x1)\, (\mathit{fwd}\, x2) \\
\mathit{bwdSub} \in (y1, y2 \in Y.t) \to Y.eq\, y1\, y2 \to X.eq\, (\mathit{bwd}\, y1)\, (\mathit{bwd}\, y2) \\
\mathit{isoEq1} \in (x \in X.t) \to X.eq\, (\mathit{bwd}\, (\mathit{fwd}\, x))\, x \\
\mathit{isoEq2} \in (y \in Y.t) \to Y.eq\, (\mathit{fwd}\, (\mathit{bwd}\, y))\, y \\
\mathit{isoLt1} \in (x1, x2 \in X.t) \to X.lt\, x1\, x2 \to Y.lt\, (\mathit{fwd}\, x1)\, (\mathit{fwd}\, x2) \\
\mathit{isoLt2} \in (y1, y2 \in Y.t) \to Y.lt\, y1\, y2 \to X.lt\, (\mathit{bwd}\, y1)\, (\mathit{bwd}\, y2)
\end{array}$$

$$\mathit{lemIdOrderIso}\ (X \in \mathit{OrderedType}) \in \mathit{OrderIso}\ X\, X$$

$$\mathit{lemIdOrderIso}\ X \equiv \begin{array}{l}\textbf{struct}\\ \left[\begin{array}{l}
\mathit{fwd} \equiv \lambda\, x \to x \\
\mathit{bwd} \equiv \mathit{fwd} \\
\mathit{fwdSub} \equiv \lambda\, x1\, x2\, h \to h \\
\mathit{bwdSub} \equiv \mathit{fwdSub} \\
\mathit{isoEq1} \equiv \lambda\, x \to X.eqRefl\, x \\
\mathit{isoEq2} \equiv \mathit{isoEq1} \\
\mathit{isoLt1} \equiv \lambda\, x1\, x2\, h \to h \\
\mathit{isoLt2} \equiv \mathit{isoLt1}
\end{array}\right.\end{array}$$

$$\mathit{lemDepOrderIso} \left(\begin{array}{l}
X \in \mathit{OrderedType} \\
Y \in X.t \to \mathit{OrderedType} \\
x1 \in X.t \\
x2 \in X.t \\
h \in X.eq\, x1\, x2
\end{array}\right) \in \mathit{OrderIso}\ (Y\, x1)\ (Y\, x2)$$

$$\begin{array}{l}
\mathit{lemDepOrderIso}\ X\, Y\, x1\, x2\, h \equiv X.eqSubst\ (\lambda\, x \to \\
\quad \mathit{OrderIso}\ (Y\, x1)\ (Y\, x)\ x1\, x2\, h)\ (\mathit{lemIdOrderIso}\ (Y\, x1))
\end{array}$$

While defining the less-than is simple, and the way to obtain both comparisons in *Prop* from their current result in *Bool* is immediate, the definition of the compare operation can't be completed. The reason for this is that the proof objects which are available after the use of the compare elimination constant are expressed in terms of the isomorphisms, while the required result is needed in the plain original types. While a (probably quite intricate) technical fix for this situation could be imagined, it clearly makes the definitions hopelessly obscure, and this is a great defect for what should be a clean and abstract formalisation of the basic database notions. We thus do not pursue the issue more in the present work.

We should just add that regardless of the technical problems involved in the comparison of fields, the definition of a tuple would consist of a finite set of fields and a proof that the fields have attribute names in the tuple's scheme, and there are no two different (A_i, x_i) and (A_i, x_i') for any A_i in the scheme. This means that every time we manipulate a tuple, we must prove that the result is a tuple over the desired result scheme. This can get tiresome and also would make the formalisation look much less clear that it should hopefully be.

Chapter 7

Conclusions

7.1 Our goals

The present work set out to do two things. First, to give a detailed and as complete as possible type-theoretic foundation for two areas: on the one hand relational programming based on allegories, and on the other hand relational database theory and its implementation. Second, we decided to use the Agda/Alfa proof system for carrying out these developments formally.

7.2 Our achievements

We will now briefly review the extent to which our goals were achieved, and what remains to be done.

7.2.1 Relational programming and allegories

We have succeeded in laying out a detailed foundation for a version of allegory theory based on constructive type theory. This version of the theory is based on the notion of an E-allegory, and we showed that the concrete allegories of constructive relations satisfy the new definitions.

Building on top of that, we then showed how to formalize various categorical constructions needed for relational programming in constructive type theory, especially relational catamorphisms. We also discussed some of the issues that arise when using relations along with functional programs seen from the type-theoretic point of view.

We constructed our foundation formally in the proof assistant Agda/Alfa, and built what amounts to a basic library of relation calculus.

7.2.2 Relational databases

We have showed several different formalizations of the relational database model in constructive type theory. In particular, we suggested a foundation for the relational model based on abstract finite sets formalized in type theory, following ideas of Filliâtre and Letouzey [29].

We then showed in detail how a data structure of lists without repetitions implements abstract finite sets (and hence database relations). Again, we carried out all these developments in the proof assistant Agda/Alfa, and built what amounts to a basic library of relational database theory.

7.2.3 Remaining work

Some of the things we aimed to do are not yet done, and give suggestions for future research. We should carry out case studies of relational programming as in Bird and de Moor's work [10] (we have done only a few examples involving the theory of lists). Also, we should construct and prove correct some efficient implementation of relational databases based on abstract finite sets (for instance, using balanced search trees or hash tables).

7.3 A discussion of our results

We have to admit that it is only when such applications to relational programming cases and the correctness of database implementations have been completed, that we will be able to definitely demonstrate the value of our work. Nevertheless, we believe that we have carried out some substantial groundwork towards achieving these goals. It is also worth remarking that the present formalization is one of the largest ones carried out in Agda/Alfa (the other maybe being Hedberg's standard library [38] and his formalization of finite automata).

As is often the case with formalization efforts, a large amount of work was invested to produce some seemingly not very revolutionary results. However, there are a number of reasons why such formalization is worth-while and has several non-routine aspects. Firstly, we have obtained some results which have been checked in detail by a mechanical (and hopefully fail-safe) tool. Like computer programs, formal proofs need to be right in every detail to be valuable when used for improving software reliability. Secondly, dependent type theories are still relatively new formalisms. The long tradition of informal reasoning of classical mathematics based on set theory is not yet there and informal reasoning is therefore less reliable. We are still discovering new aspects of the theory. This thesis explores an application of type theory which has not previously been studied much. In particular, we explore constructive relation calculus, and although it shares many features with classical relation calculi we have discovered some new aspects, for example, the notions of E-relation and E-allegory. Having realized that E-allegories are a suitable notion of allegory for type theory is a key insight. Another is that you only get power objects by restricting E-allegories to the finite decidable case. Yet another one is that the use of power allegories does not seem essential when relational programming is based on allegories. Relational catamorphisms can be constructed directly in constructive type theory.

Regarding the results on relational databases, we think that the idea that dependent types are very relevant for formalizing database schemes is a valid one, but to carry it out in practice is non-trivial. The interactions between issues of computability and expressibility are interesting and show some genuine difficulties. It is difficult to capture the full extent of the idea of relation scheme. There are genuinely interesting issues having to do with the presence of proof objects in constructions and certain aspects of intensional type theory. On a different vein, we have established a link between abstract relational structures and concrete implementations in the setting of type theory which is quite encouraging. This should provide a firm set-up for a detailed investigation of how to relate abstract set-theoretic relations with implementations in real programming languages, an issue which to our knowledge has not been dealt with before in the type-theoretic literature.

7.4 Regarding the tools used for this work

The tool Agda/Alfa is still experimental. It is the latest in a succession of proof systems for dependent type theory developed in Gothenburg. Many aspects of these proof systems are still in a state of flux, ranging from the exact rules of the base logic to implementation details, and interface design. In particular, the Agda/Alfa system we used does not provide assistance for automatic proof search, so proofs had to be constructed manually in complete detail.

Since it is an experimental tool, we found some aspects of it rather non-intuitive and as a consequence it slowed us down. Nevertheless, it should probably be said that the complexity of any such big formalization project is beyond the ability of a single average formalizer, just as the complexity of any big non-trivial piece of software is beyond the coding capability of any single average programmer. The application of additional tools used in similar ways as software engineers do when faced with big challenges seems like a good idea and good advice to anyone attempting future formalizations of non-trivial scope. The proof assistant could be made more suitable for large formalizations with the addition of more libraries, enhanced documentation, more powerful pretty-printing facilities, and automation plugins which could automatically find simple proofs. It should be expected that the ongoing development of new variants and plugins for Agda/Alfa will help to automate some of the more tedious and time-consuming parts of big formalization efforts in the near future.

7.5 Brief discussion of future work

As we already mentioned in the introduction of this thesis, we expect that the proof code developed during this work will be useful as a basis for developing new interesting results. In particular, some more realistic relational program derivations that can exemplify the ways in which the relational and the type-theoretic approaches can work in unison should be built as a next step.

We also mentioned before that often too much time was required to get Agda/Alfa proofs of results that, while necessary and interesting by themselves, had proofs that weren't actually interesting at all. Recently, a tool for automatic proof search (called Agsy) for Alfa/Agda has been implemented by F. Lindblad. A new version of Agda (called Agda Light) with a prover plugin is being implemented by A. Abel and U. Norell, and there is also a FOL prover plugin for Agda by Ikegami Daisuke. Details of all these recent developments can be found in [71]. These new tools and enhancements to the proof assistant should mean that when we continue with our formalization efforts more of our time can be freed from such uninteresting proofs and spent instead on the really rewarding ones. This will be particularly important when it comes to proving correctness of a database implementation using the kind of data-structures which are used in actual relational database managers.

Bibliography

[1] H. Andréka, I. Németi and I. Sain: Algebraic Logic. In D. Gabbay and F. Guenthner (eds.), *Handbook of Philosophical Logic, Second Edition, Volume 2*. Kluwer Academic Publishers, 2001.

[2] L. Augustsson: Cayenne - a language with dependent types. *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming, Baltimore, Maryland, United States, September 26-29, 1998*, pp. 239–50. ACM Press, 1998.

[3] H. Barendregt and H. Geuvers: Proof-assistants using Dependent Type Systems. In A. Robinson and A. Voronkov (eds.): *Handbook of Automated Reasoning*, Elsevier Science Publishers B.V., 2001, Volume II, chapter 18.

[4] R.S.M. de Barros: On the Formal Specification and Derivation of Relational Database Applications. Electronic Notes in Theoretical Computer Science, Vol. 7, 1998.

[5] G. Barthe, V. Capretta and O. Pons: Setoids in type theory. Journal of Functional Programming, Volume 13, Issue 2, March 2003, pp. 261-293.

[6] G. Barthe, G. Dufay, M. Huisman, and S. Melo de Sousa: Jakarta: A Toolset for Reasoning about JavaCard. In I. Attali and T. Jensen (eds.), *Smart Card Programming and Security - International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001 - Proceedings*. LNCS 2140, Springer-Verlag, 2002, pp. 2–18, 2001.

[7] R. Berghammer, T. Hoffmann, B. Leoniuk and U. Milanese: Prototyping and Programming with Relations. Electronic Notes in Theoretical Computer Science, Volume 44, Issue 3, May 2003.

[8] R. Berghammer, B. Möller and G. Struth (eds.): *Relational and Kleene-Algebraic Methods in Computer Science*, 7th International Seminar on Relational Methods in Computer Science and 2nd International Workshop on Applications of Kleene Algebra, Bad Malente, Germany, May 2003, Revised Selected Papers. LNCS 3051, Springer-Verlag, 2004.

[9] I. Beylin and P. Dybjer: Extracting a proof of coherence for monoidal categories from a formal proof of normalization for monoids. In S. Berardi and M. Coppo (eds.), *Types for Proofs and Programs - International Workshop, TYPES '95, Torino, Italy, June 5 - 8, 1995 Selected Papers*. LNCS 1158, Springer Verlag, 1996, pp. 47–61.

[10] R. Bird and O. de Moor: *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1997.

[11] A. Bove: *General Recursion in Type Theory*. PhD thesis, Department of Computing Science, Chalmers University of Technology and Göteborg University, Sweden, 2002.

[12] C. Brink, W. Kahl and G. Schmidt (eds.): *Relational Methods in Computer Science*. Advances in Computer Science. Springer-Verlag, 1997.

[13] Brown, C., Hutton, G.: Categories, Allegories and Circuit Design. In *Proceedings of the 10th Annual IEEE symposium on Logic in Computer Science, IEEE Computer Society Press, Los Alamitos, California, July 1994*. IEEE, 1994.

[14] P. Buneman, S. Naqvi, V. Tannen and L. Wong: Principles of programming with complex objects and collection types. Theoretical Computer Science, Vol. 149, pp. 3–48.

[15] A. Carboni and R.F.C. Walters: Cartesian Bicategories I. Journal of Pure and Applied Algebra, Vol. 49, 1987, pp. 11–32.

[16] D.M. Cattrall, *The Design and Implementation of a Relational Programming System*. PhD Thesis, Department of Computer Science, University of York, England, 2002.

[17] E.F. Codd: A Relational Model of Data for Large Shared Data Banks. Communications of the ACM, Volume 13, Number 6, pp. 377–387, June 1970.

[18] C. Coquand: Agda home page. `http://www.cs.chalmers.se/~catarina/agda/`

[19] C. Coquand and T. Coquand: Structured Type Theory. In *Workshop on Logical Frameworks and Meta-languages, Paris, France, September 1999, Proceedings*. Part of PLI'99 (Colloquium on Principles, Logics, and Implementations of High-Level Programming Languages).

[20] J. Dallien: RelDT: Relational Dual Tableaux Automated Theorem Prover. `http://logic.stfx.ca/reldt/`.

[21] H. Darwen and C.J. Date: The Third Manifesto. ACM SIGMOD Record, Volume 24, Issue 1 (March 1995), pp. 39–49.

[22] J.E. Dawson and R. Goré: Embedding Display Calculi into Logical Frameworks: Comparing Twelf and Isabelle. Electronic Notes in Theoretical Computer Science, Vol. 42, 2001.

[23] J. E. Dawson and R. Goré: A Mechanised Proof System for Relation Algebra using Display Logic. In J. Dix, L. Farías del Cerro, U. Furbach (eds.): *Logics in Artificial Intelligence: European Workshop, JELIA'98, Dagstuhl, Germany, October 1998. Proceedings*, LNCS 1489, Springer-Verlag, 1998, pp. 264–278.

[24] B. Dwyer: LIBRA: A Lazy Interpreter of Binary Relational Algebra. Computer Science Technical Report 95-10, Department of Computer Science, University of Adelaide, Australia, 1995.

[25] P. Dybjer: A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. Journal of Symbolic Logic, Vol. 65, no. 2, June 2000, pp. 525–549.

[26] P. Dybjer: Inductive Families. Formal Aspects of Computing, Vol. 6, 1994, pp. 440–465.

[27] P. Dybjer and V. Gaspes: Implementing a category of sets in ALF, manuscript, 1993.

[28] P. Dybjer and A. Setzer: Indexed induction-recursion. In *Proof Theory in Computer Science - International Seminar, PTCS 2001 Dagstuhl Castle, Germany, October 7-12, 2001*, pp. 93–113.

[29] J.-C. Filliâtre and P. Letouzey: Functors for Proofs and Programs. In D.Schmidt, *Programming Languages and Systems: 13th European Symposium on Programming, ESOP 2004, Barcelona, Spain, March 29 - April 2, 2004. Proceedings*, LNCS 2986, Springer-Verlag, 2004, pp. 370–384.

[30] A. Formisano, E.G. Omodeo and M. Temperini: Goals and Benchmarks for Automated Map Reasoning. Journal of Symbolic Computation, Vol. 29, 2000, pp. 259–297.

[31] P.J. Freyd and A. Scedrov: *Categories, Allegories.* North-Holland Mathematical Library, Volume 39. Elsevier Science Publishers B.V., 1990.

[32] Furusawa, H., Kahl, W.: A Study on Symmetric Quotients. Bericht Nr. 1998-06, Universität der Bundeswehr München, Fakultät für Informatik, December 1998.

[33] C. Gonzalía: Towards a Formalisation of Relational Database Theory in Constructive Type Theory. In [8], pp .137–148.

[34] C. Gonzalía: The Allegory of E-Relations in Constructive Type Theory. In J. Desharnais, M. Frappier and W. MacCaull (eds.), *Relational Methods in Computer Science - The Québec Seminar*, pp. 19–38. Proceedings Series, Vol. 1. Methoδos Verlag, 2002.

[35] C. Gonzalía: *Relation Calculus in Martin-Löf Type Theory*. Licenciate thesis, Department of Computing Science, Chalmers University of Technology and Göteborg University, 2002.

[36] T. Hallgren: Alfa home page. `http://www.cs.chalmers.se/~hallgren/Alfa/`

[37] C. Hattensperger, R. Berghammer and G. Schmidt: RALF - A Relation-Algebraic Formula Manipulation System and Proof Checker. In M. Nivat el al. (eds.): *Algebraic Methodology and Software Technology (AMAST '93), Proceedings of the Third International Conference on Methodology and Software Technology, University of Twente, Enschede, The Netherlands, 21-25 June, 1993*, Springer-Verlag, 1993.

[38] M. Hedberg: New standard library for Alfa. `http://www.cs.chalmers.se/pub/users/hallgren/Alfa/Alfa/Library/New/`

[39] Hofmann, M.: *Extensional concepts in intensional type theory*. PhD thesis CST-117-95, Department of Computer Science, The University of Edinburgh, July 1995.

[40] R. Holt: Introduction to the Grok Programming Language. 2002.

[41] G. Huet and A. Saïbi: Constructive Category Theory. In *Proceedings of the joint CLICS-TYPES Workshop on Categories and Type Theory, Goteborg (Sweden), January 1995.*

[42] Hustadt, U., Schmidt, R. A.: MSPASS: Modal Reasoning by Translation and First-Order Resolution. In Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2000), LNAI Vol. 1847, pp. 67-71. Springer Verlag, 2000.

[43] Jones, G., Sheeran, M.: Circuit Design in Ruby. In *Formal Methods for VLSI Design*, ed. J. Staunstrup, North Holland, 1990.

[44] W. Kahl: Calculational Relation-Algebraic Proofs in Isabelle/Isar. In [8], pp. 178–190.

[45] W. Kahl: Refactoring Heterogeneous Relation Algebras around Ordered Categories and Converse. In Journal on Relational Methods in Computer Science, Vol. 1, pp. 277–313, 2004.

[46] W. Kahl and G. Schmidt: Exploring (Finite) Relation Algebras Using Tools Written in Haskell. Technical Report Nr. 2000-02, Fakultät für Informatik, Universität der Bundeswehr München, October 2000.

[47] Y. Kinoshita: A Bicategorical Analysis of E-Categories, Mathematica Japonica, Vol. 47, No.1, pp. 157–169, 1998.

[48] P. Knijnenburg and F. Nordemann: Two Categories of Relations. Technical Report no. 94-32, Dept. of Computer Science, Leiden University, the Netherlands, 1994.

[49] G. Lee, R. Little, W. MacCaull and B. Spenser: ReVAT - Relational Validation by Analytic Tableaux. Draft, Logic and Computation Group, St. Francis Xavier University, Canada.

[50] W. MacCaull and E.Orlowska: A Calculus of Typed Relations. In [8], pp. 191–201.

[51] M.E. Maietti and S. Valentini: Can you add power-sets to Martin-Löf's intuitionistic set theory?, Mathematical Logic Quarterly, vol. 45, 1999, pp. 521–532.

[52] P. Mäenpää and M. Tikkanen: Dependent Types in Databases. Workshop on Dependent Types in Programming, ESPRIT Working Groups APPSEM and TYPES. Göteborg, Sweden, 27-28 March, 1999.

[53] D. Maier: *The Theory of Relational Databases.* Computer Software Engineering Series. Computer Science Press, 1983.

[54] B. Nordström, K. Petersson and J.M. Smith: *Programming in Martin-Löf's Type Theory. An Introduction.* International Series of Monographs on Computer Science, No. 7. Oxford University Press, 1990.

[55] D. von Oheimb and T. Gritzner: RALL - Machine-supported proofs for Relation Algebra. In W. McCune (ed.): *Automated Deduction - CADE-14, 14th International Conference on Automated Deduction, Townsville, North Queensland, Australia, July 13-17, 1997, Proceedings.* LNCS 1249, pp. 380-394, 1997. Springer-Verlag.

[56] C. Okasaki: *Purely Functional Data Structures.* Cambridge University Press, 1998.

[57] H. Okuma and Y. Kawahara: Relational Aspects of Relational Database Dependencies. Bulletin of Informatics and Cybernetics, Research Association of Statistical Sciences, Vol. 32, No. 2, December 2000, pp. 93–104.

[58] B.I. Plotkin: *Universal Algebra, Algebraic Logic and Databases.* Kluwer Academic Publishers, 1993.

[59] Qiao Haiyan: *Testing and Proving in Dependent Type Theory*. PhD thesis, Department of Computing Science, Chalmers University of Technology and Göteborg University, 2003.

[60] P. Rajagopalan: *A Type Theory Approach to the Specification and Synthesis of Database Models.* PhD Thesis. Department of Computer Science, University of Western Australia, Crawley, Australia, December 1993.

[61] P. Rajagopalan and C.P. Tsang: A Generic Algebra for Data Collections Based on Constructive Logic. In V.S. Alagar and M. Nivat (eds.), *Algebraic Methodology and Software Technology, 4th International Conference, AMAST '95, Montreal, Canada, July 3-7, 1995, Proceedings*, pp. 546–560. Lecture Notes in Computer Science, Volume 936. Springer-Verlag, 1995.

[62] O. Rasmussen: Formalising Ruby in Isabelle ZF. In Proceedings of the First Isabelle Users Workshop, Tech. Report 379, pages 246–265. University of Cambridge, September 1995.

[63] D. Scott: Domains for Denotational Semantics. In M. Nielsen and E.Schmidt *Proceedings 9th International Colloquium on Automata, Languages and Programming (ICALP '82), Aarhus, Denmark, July 12-16, 1982*. LNCS 140, pp. 577–613. Springer-Verlag, 1982.

[64] G. Schmidt and T. Ströhlein: *Relations and Graphs. Discrete Mathematics for Computer Scientists.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1993.

[65] C. Sinz: System Description: ARA - An Automatic Theorem Prover for Relation Algebras. In D. McAllester (ed.): *Automated Deduction CADE-17, 17th International Conference on Automated Deduction Pittsburgh, PA, USA, June 17-20, 2000, Proceedings*. LNAI 1831, pp. 177–182, Springer-Verlag, 2000.

[66] L. Sterling and E. Shapiro: *The Art of Prolog, Second Edition - Advanced Programming Techniques*. The MIT Press, 1994.

[67] A. Tarski: On the Calculus of Relations. The Journal of Symbolic Logic, Vol. 6, No. 3, September 1941.

[68] M. Vaccari: *Calculational Derivation of Circuits.* PhD Thesis, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, 1998.

[69] S. Valentini: Extensionality Versus Constructivity. Mathematical Logic Quarterly, Vol. 48, no. 2, 2002, pp. 179–187.

[70] J. Van den Bussche: Applications of Alfred Tarski's Ideas in Database Theory. In L. Fribourg (ed.), *Computer Science Logic: 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, pp. 20–37. Lecture Notes in Computer Science, Volume 2142. Springer-Verlag, 2001.

[71] Various: Proceedings of the Agda Implementor's Meeting III. `http://coverproject.org/AgdaPage/AIM3/`

[72] Various: Cover Translator. `http://coverproject.org/CoverTranslator/`