Small Model Theorems for Data Independent Systems in Alloy

Lee Momtahan
Oxford University Computing Laboratory

December 21, 2007

Abstract

A system is data independent in a type T if the only operations allowed on variables of type T are input, output, assignment and equality testing. This property can be exploited to give procedures for the automatic verification of such systems independently of the instance of the type T.

Alloy is an extension of first-order logic for modelling software systems. Alloy has a fully automatic analyzer which attempts to refute Alloy formulas by searching for counterexamples within a finite scope. However, failure to find a counterexample does not prove the formula correct.

A small model theorem is a theorem which shows that if a formula has a model then it has a model within some finite scope. The contribution of this thesis is to give a small model theorem which applies when modelling data-independent systems in Alloy.

The theorem allows one to detect automatically whether an Alloy formula is data independent in some type T and then calculate a threshold scope for T, thereby completing the analysis of the automatic analyzer with respect to the type T.

We derive the small model theorem using a model-theoretic approach. We build on the standard semantics of the Alloy language and introduce a more abstract interpretation of formulas, by way of a Galois insertion. This more abstract interpretation gives the same truth value as the original interpretation for many formulas. Indeed we show that this property holds for any formula built with a limited set of language constructors which we call data-independent constructors. The more abstract interpretation is designed so that it often lies within a finite scope and we can calculate whether this is the case and exactly how big the finite scope need be from the types of the free variables in the formula. In this way we can show that if a formula has any instance or counterexample at all then it has one within a threshold scope, the size of which we can calculate.

Acknowledgements

I am very grateful to my supervisor Bill Roscoe for the help he has given to me in producing this thesis. I would like to thank Michael Goldsmith and Jeremy Gibbons for the useful comments they gave during my confirmation. Thanks also go to Michael for being my examiner along with Michael Huth. Gavin Lowe and Ranko Lazić gave me useful comments when the thesis was at an early stage.

Andrew Martin's assistance in using a type checker in the thesis has undoubtedly saved me a great deal of time and errors. I would also like to thank Andrew Simpson and David Gavaghan for continuing to employ me while finishing the thesis.

Thanks go to Daniel Jackson for supporting the idea of the thesis and clarifying some of the intricacies of the Alloy type system. Finally I must thank Geoff Williams, Doug Creager, David Faitelson, Charlie Chricton and David Power for their friendship and support.

Contents

1 Introduction **4**
- 1.1 Computer-aided verification . . . 4
- 1.2 Contribution of the thesis . . . 8
- 1.3 Overview . . . 9
- 1.4 Notation . . . 10

2 Language definition **11**
- 2.1 Types . . . 11
- 2.2 Variables . . . 18
- 2.3 Formulas . . . 20
- 2.4 Scopes . . . 27

3 Semantic data independence **30**
- 3.1 Quotient bindings . . . 30
- 3.2 Galois insertion . . . 31
- 3.3 The quotient of a binding . . . 37
- 3.4 Semantic data independence . . . 42

4 Syntactic data independence **45**
- 4.1 Data-independent constructors . . . 45
- 4.2 Data-dependent constructors . . . 61

5 Thresholds **64**
- 5.1 Maximum data relations . . . 64
- 5.2 Folds and equivalence relations . . . 68
- 5.3 Calculation of thresholds . . . 71
- 5.4 Small model theorem . . . 100
- 5.5 Necessity of thresholds . . . 103

6 Enhancements **108**
- 6.1 Converting to negation normal form and Skolemizing . . . 108
- 6.2 Discounting use of type variables as expressions . . . 109
- 6.3 Rewriting fixed arity types . . . 110

6.4 Eliminating variables 110

7 Multiple data independent type variables **112**
7.1 Applying the small model theorem repeatedly 112
7.2 Extending the small model theorem 114

8 Strong data independence **139**
8.1 A smaller language 139
8.2 Strongly-data-independent formulas 140

9 Case study **145**
9.1 Hotel room locking 145

10 Conclusions and further work **156**
10.1 Conclusions 156
10.2 Further work 158

A Proofs **159**

B Notation **172**
B.1 Quantification 172
B.2 Set comprehension 172
B.3 Symbols 173

Chapter 1

Introduction

Computer systems play an increasingly significant role in our prosperity, yet at the same time they are becoming increasingly large and complex, and more prone to failure. Failure of computer systems can cause a significant economic impact and even loss of life [LT93].

Correctness of such systems is often verified informally and much emphasis is placed on testing. An alternative approach to verifying correctness, called *formal methods* is based on precise and rigorous mathematical reasoning. Such methods can help to give much greater confidence in the correctness of a design.

The formal methods approach to software engineering proceeds with precise mathematical models of both how the system should behave (*specification*) and how the system is designed (*implementation*). Verifying the implementation meets the specification becomes a question which can be argued beyond mathematical doubt.

Unfortunately a naïve formal methods approach generally requires a high degree of expensive labour to construct the necessary mathematical proofs. Therefore a number of software tools have been, and continue to be, developed to automate the formal methods approach. Research (and use of) such tools falls under the banner of *Computer-aided verification*.

1.1 Computer-aided verification

In this section we introduce the related work from the area of Computer-aided verification.

1.1.1 Theorem proving

With theorem proving the specification and implementation are represented in formalisms which allow the satisfaction of the specification by the implementation to be represented as a logic formula; then axioms and inference rules are used to prove the formula. Given the size and complexity of today's systems,

often such proofs can be long and complex. But unlike proof in mathematics the usual social process of review by independent peers is not available or too expensive [DLP79]. Therefore a number of proof tools have been created to check proofs e.g. PVS [ORS92], HOL [GM93]. Moreover, theorem proving tools can make parts of the proof construction automatic, allowing users to concentrate on more creative aspects of the proof, such as choosing inductive hypotheses.

Advantages and disadvantages

Theorem proving is a very general approach and can be applied to a wide variety of systems, in particular *infinite-state* and *parameterized* systems (see Section 1.1.4). The main disadvantage with theorem proving is that a high degree of highly skilled labour is required, even when proof tools are used.

1.1.2 Model checking

This is a brute-force approach to verifying correctness, where traditionally a complete state-transition graph of the implementation is constructed and automatically checked state by state that it satisfies its specification [EMGP99]. SPIN [Hol90] and FDR [FSEL92] are model checkers, for example. A more efficient approach taken by the SMV model checker is to represent the state-transition graph as a binary decision diagram [BCM$^+$92].

When non-deterministic models are used a property known as *fairness* can become important. Informally speaking, a computation can be considered fair if, during its execution, every event which becomes possible infinitely often is realized infinitely often, otherwise it is unfair. Consider for example a lossy transmission line where the emission of a message give rise to reception of a message or the loss of a message. A functioning where an infinity of messages are emitted while only a finite number are received is unfair. A formal definition of fairness is given in [QS83].

Advantages and disadvantages

The main advantage with this approach is that it is completely automatic. Also, if an implementation does not meet its specification it is possible to generate a counterexample showing an execution path that leads to the error which can help in fixing the problem. However, in general one can check a complete state-transition graph in a finite amount of time only when the system is *finite state* (but see Sec. 1.1.4 for how this limitation can be overcome in specific cases). Furthermore, some systems suffer from *state explosion*, which means that although the system is finite state, the size of its transition graph is so large as to be practically uncheckable using a reasonable amount of time and computational resource.

1.1.3 Model finding

Like theorem proving, the refinement of the specification by the implementation is first represented as a logic formula. An attempt to refute the logic formula is made by automatically searching for a counterexample.

Model finding is similar to model checking since both methods are fully automated. However, whereas model checkers take a specification, usually in the form of a temporal logic formula, and check whether an implementation is a model of it, model finders take a logic formula and attempt to find a model of it.

Advantages and disadvantages

The advantages and disadvantages of model finding are very similar to model checking. In order to keep the search for a counterexample bounded it is necessary to give a *scope* to the model finder. For each type used in the formula, the scope bounds the number of elements used to instantiate the type in the counterexample. Therefore it is only possible to check finite-state systems completely.

1.1.4 Infinite and parameterized verification

In the context of model checking a system is *finite state* if and only if its state-transition graph has a finite number of nodes. Complete model checking of general infinite-state systems is not possible in a finite amount of time. A related problem to the verification of infinite-state systems is the verification of parameterized systems where the value of a parameter ranges over an infinite set, e.g. checking deadlock freedom of a token ring network for an arbitrary number of nodes in the network.

In the context of model finding, infinite and infinitely parameterized systems are ones which require checks where it is not possible to give a bound on the size of types. This is clearly an equivalent concept since a system with an infinite number of states can not be represented by a finite number of variables declared using finite constructs on finite types, whereas a system with a finite number of states can.

It is impossible to build a model checker or finder which can in general deal with infinite-state systems or the general parameterized verification problem since this would solve a more specific problem: the *Halting Problem* which is known to be insoluble [Tur37].

Nevertheless an active research area involves finding techniques which allow special cases of these problems which can be dealt with. For example the method of invisible auxiliary assertions and the method of counter abstraction [PZ03]. These techniques can not only reduce the state space from infinite to finite, but, for some problems, can reduce state-explosion.

Data independence

One special case of infinite and parameterized verification is called *data independence* [Wol86] [LN00]. Informally, a system is *data independent* with respect to a type T if it can only input, output and store values of this type as well as copy them within its variables. The control-flow of such a system is not affected by different values; changing the input data will not affect behaviour except for the corresponding output data. Because the control-flow is independent of the type used this can be exploited in the verification of such systems.

These strict conditions on data independence can often be relaxed to allow equality testing between variables of the type (known as *weak* data independence), and uninterpreted constants and finite range functions on the type as well, while still maintaining decidability results.

Data-independent systems are very common, for example a communication protocol is usually data independent in the type communicated. Memory or database systems may be data independent with respect to the type of values which they store as well as the type of address.

Data-independent systems which operate on arrays where the source and/or target type of the arrays are data independent have also been researched. Classes of systems, depending on the type and number of arrays used as well as the exact array operations allowed, have been defined for which decidability results hold [New03].

One way to check data-independent systems is through finite-instantiation methods. Threshold theorems can be developed which show that once a system is verified for all sizes of its data-independent type variable up to a particular value, then the system is correct for all non-empty instantiations of the type [Wol86].

1.1.5 Alloy and the Alloy Analyzer

The Alloy [Jac02a] [Jac02b] [Jac06] modeling language consists of first order logic with sets and relations. It is roughly a subset of the Z notation [Spi92], and also has similarities with UML's OCL [RJB99][WK99]. Alloy is designed to bring to Z-style specification the kind of automation offered by model checkers.

The Alloy Analyzer is a model finder for the Alloy language. The Analyzer works by first translating formulas in the Alloy language into a smaller Alloy Kernel language [Jac02a]. Because the Analyzer is given a finite *scope* (see Section 1.1.3), it can transform an Alloy Kernel language formula into a boolean formula, such that the boolean formula has a model exactly when the Alloy Kernel language formula (and hence the original formula) has a model within the given scope [Jac00]. To test the boolean formula, the Analyzer wraps off-the-shelf SAT solvers, such as SATO [Zha97] or RelSAT [BS97].

1.1.6 MACE and other model finders

Other model finders such as MACE [McC01] and FINDER [Sla94] are oriented toward solving problems in mathematics e.g. finite algebras, rather than software engineering. These tools find models in languages which are untyped; they take as their scope a single natural number which determines the size of domain to search.

1.2 Contribution of the thesis

In this section we explain the contribution of this thesis.

Small model theorems

A *small model theorem* is a generic name for a theorem which proves that in some language a formula has a model only if it has a model within some finite scope (see Section 1.1.3). In [Jac02a], Jackson asks if a small model theorem could be found for Alloy:

> "By its very nature, the analysis is not *complete*: a failure to find a counterexample does not prove a theorem correct. It may be possible to perform a static analysis that establishes a 'small model theorem'. If one can show that a formula has a model only if it has a model within a given scope, an analysis within that scope would constitute a proof. Because Alloy is based on traditional logic and set notions, it may be possible to develop such a technique by applying known results from model theory."

Clearly the notion of small model theorem and the finite instantiation method for data independence are related, even though the term data independence is used in the context of model checking. It seems reasonable to expect the existence of a small model theorem which applies to a general class of systems: those viewed as data independent from the model checking perspective. The theory which will be developed in this thesis is inspired by this reasoning and has similarities with data-independence theory in model checking, in particular Lazić's semantic study of data independence [LN00]. Although our approach is model theoretic, we are not aware of known results from model theory which could be directly applied.

Contribution of the thesis

The contribution of this thesis is to give a *small model theorem (SMT)*, which applies when modeling data-independent systems in the particular context of Alloy. This is completely novel work. The author's original publication presenting his early findings in this area is [Mom04].

Given a formula and a type variable used within that formula the SMT allows one to determine whether the type variable is a data-independent one.

If so, the SMT then gives a threshold on the size of this data-independent type variable, as a function of the size of the other type variables in the formula. Taken together these sizes give a threshold scope and the SMT guarantees that if the formula has no counterexamples at this scope then increasing the size of the data-independent type variable still yields no counterexamples.

If *a priori* bounds are known for all the type variables in a formula, except one data-independent one, the SMT can be used to complete the analysis performed by the Alloy Analyzer. Even if *a priori* bounds are not known, the SMT can be considered advantageous in that it removes the guesswork in choosing a size for one of the type variables in the formula.

A further contribution of the thesis is to extend the SMT to deal with some formulas which have more than one data-independent type variable. Here the SMT gives a threshold size for each of these type variables (again as a function of the size of the other type variables in the formula). The SMT guarantees that if no counterexamples are found at the threshold scope, then increasing the size of any of the data-independent type variables still yields no counterexamples.

The reason the SMT was developed for the Alloy language in particular is because it is a typed language with a model finder implementation available. A typed language is required for there to be a notion of type variable and hence data-independent type variable, and a model finder implementation is required for the work of the thesis to have practical as well as theoretical benefits.

1.3 Overview

In Chapter 2 we give a formal description of the language which will be used for the SMT. This language is almost the same as the Alloy Kernel language (version 2.0) [Jac02a] [Jac06, Appendix C], and one can easily translate formulas form the full-blown language into this language. This chapter proves a foundation for the thesis using the work of Jackson. The remainder of the thesis is original work.

We then start working towards the SMT in Chapter 3 by defining a semantic notion of data independence. Formulas which satisfy this definition have a useful property in that we can evaluate their truth under some binding by instead choosing a *quotient* binding. The advantage of the *quotient* binding is that it usually lies within a smaller scope.

However, although the semantic definition is of theoretical importance, it has limited practical value, because we have no way of calculating whether a formula satisfies this definition. Therefore in Chapter 4 we define a syntactic notion of data independence. The problem of determining whether a formula satisfies this definition is decidable. We show that a formula which satisfies the syntactic definition also satisfies the semantic definition. The converse does not hold: it is possible to write a formula which satisfies the semantic definition without satisfying the syntactic one, but such formulas are convoluted and are unlikely to arise in practice.

In Chapter 5 we consider whether and how we can choose quotient bindings

such that they lie within a small scope, exactly how large this scope needs to be, and draw on the work of the previous chapters to give a small model theorem.

The following short chapter gives some techniques which can be used to enhance the SMT. For example procedures to rewrite formulas so that they can be made to satisfy the syntactic definition of data independence, and procedures to rewrite type expressions in order to reduce the size of thresholds generated by the SMT.

At this point of the thesis we will have considered only a single data-independent type variable. Chapter 7 considers some of the shortcomings when trying to apply the theory to formulas with multiple data-independent type variables. We show how our theory can be extended to deal with multiple data-independent type variables in a number of situations.

In Chapter 8 we look at a special case of data independence known as *strong* data independence. Based on intuition and data-independence work in model-checking (e.g. [Laz99]) we expect a threshold of 2 to be sufficient to check a strongly-data-independent type variable. We do not manage to obtain such a threshold, although we do give an indication as to why this a difficult task.

In Chapter 9 we give an example problem and show how the SMT can be applied. Chapter 10 concludes the thesis by summarizing the contribution and indicating potential areas of future work.

There are also two appendices. The proofs of some of the theorems from the main text are placed in the first. Some notes on notation are provided in the second.

1.4 Notation

The notation i.e. the meta-language used in the thesis is based on the Z notation [Spi92] [WD96]. All formulas in the thesis use this notation. We deviate from the Z standard in the declaration of base types and constants, adopting a more mathematical style. For free types we use a BNF style syntax.

To aid the reader we differentiate constructs of the language using a fixed-width typeface e.g. `language` and constructs of the meta-language using a proportionally spaced typeface e.g. *meta*.

Every formula in this thesis, except those appearing in examples, has been checked with the fuzz [Spi00] type-checker.

Chapter 2

Language definition

In this chapter we define the modeling language which will be used for the small model theorem.

The language is based on the Alloy Kernel language [Jac06, Appendix C], into which problems from the full-blown Alloy language can easily be translated. Indeed the first step that the Alloy Analyzer takes in analyzing a problem is to translate it into the Alloy Kernel language. The Alloy Kernel language is a smaller language and consequently reduces the size of the small model theorem we shall develop. Nevertheless it contains all the elements relevant to our concerns.

2.1 Types

In this section we define the syntax and semantics of types. We start with a set of primitive entities used in modelling called *atoms*. They are indivisible, immutable and uninterpreted [Jac06, pp. 35].

Definition 2.1 Atoms

Let $Atom$ denote a set whose elements are called atoms. We use the symbols x, y and z to refer to atoms. ◇

A *value* can be used to represent a particular atom, a set of atoms, or a relation between atoms.

Definition 2.2 Values

In the context of a set of atoms, a value is a set of finite sequences of atoms. The set of values is thus:

$$Value = \mathbb{P}(\operatorname{seq} Atom)$$

We will use the symbols xs, ys and zs to refer to sequences of atoms, and V and W to refer to values. ◇

Later, when we define the type system of the language, we will see that the sequences contained in a value are all of the same length: the arity of the relation the value represents. This will be shown in Lemma 2.17.

Example 2.3
Suppose $Atom = \{hydrogen, helium, lithium\}$. Then the value $\{\langle hydrogen \rangle\}$ represents an atom (the lightest). The value $\{\langle hydrogen \rangle, \langle lithium \rangle\}$ represents a set of atoms (those which are reactive). The value $\{\langle hydrogen, helium \rangle, \langle helium, lithium \rangle, \langle hydrogen, lithium \rangle\}$ represents a relation (the order by weight). ◇

Remark 2.4 We emphasise the difference between an object and its representation in our language since this gives rise to some unconventional equalities. In particular an object and the singleton set containing that object are both represented as the same value. Furthermore, although values can represent flat relations they cannot represent higher order relations i.e. relations between relations. Alloy is designed in this way to make the analysis of formulas easier, and since our language is based on the Alloy Kernel language we do the same. See [Jac06, Section 3.2.2] for further details on the rationale behind this aspect of Alloy's design. ◇

We use *type variables* to model the different kinds of entities that occur when modeling.

Definition 2.5 Type variables
A set of names for type variables is denoted $TypeVar$. We will use the letters X, Y, Z to refer to elements of this set. ◇

We give the meaning of a type variable in terms of atoms by using a *set map*.

Definition 2.6 Set maps
In the context of a set of atoms, a set map is a partial map from type variables to sets of atoms. Distinct type variables are mapped to disjoint sets.

$$\begin{aligned} &SetMap = \{\delta : TypeVar \pfun \mathbb{P}\, Atom \mid \\ &\quad \forall X, Y : \operatorname{dom} \delta \bullet \\ &\quad X \neq Y \Rightarrow \delta(X) \cap \delta(Y) = \emptyset\} \end{aligned}$$

The image of a type variable under a set map is called a carrier set. We will use the symbol δ to stand for a set map. ◇

Example 2.7
Let:

$$\begin{aligned} &Atom = \{bill, ben, concurrency, jack, john, jim, topology\} \\ &TypeVar = \{\texttt{Professor}, \texttt{Student}, \texttt{Subject}, \texttt{City}\} \end{aligned}$$

and:

$$\begin{array}{l}\delta = \{ \\ \quad \texttt{Professor} \mapsto \{bill, ben\}, \\ \quad \texttt{Student} \mapsto \{jack, john\}, \\ \quad \texttt{Subject} \mapsto \{concurrency, topology\} \\ \}\end{array}$$

Then δ is a set map. ◇

We now begin to define the syntax of types beginning with the *relational types*.

Definition 2.8 Relational types
We define the set of relational types as follows:

$$\begin{array}{rcl} \mathit{RelationalType} ::= & & \mathit{TypeVar} \\ & | & \mathit{RelationalType} \texttt{ + } \mathit{RelationalType} \\ & | & \mathit{RelationalType} \texttt{ -> } \mathit{RelationalType} \end{array}$$

The type P + Q represents the union of the types P and Q, and the type P -> Q represents the Cartesian product of the types P and Q. We will use the letters P and Q to stand for relational types. ◇

The following function will be used shortly in defining the semantics of relational types.

Definition 2.9 Used type variables
We define a function *Used*, which applies to relational types and returns the set of type variables which are used:

$$\begin{array}{l} \mathit{Used}(X) = \{X\} \\ \mathit{Used}(P \texttt{ + } Q) = \mathit{Used}(P) \cup \mathit{Used}(Q) \\ \mathit{Used}(P \texttt{ -> } Q) = \mathit{Used}(P) \cup \mathit{Used}(Q) \end{array}$$

◇

Note that in the above definition X is universally quantified over type variables, and P and Q are also universally quantified over relational types. We will adopt the convention throughout the thesis that free variables of the metalanguage are universally quantified.

Definition 2.10 Preliminary semantics of relational types
Given $P : \mathit{RelationalType}$ and $\delta : \mathit{SetMap}$ such that $\mathit{Used}(P) \subseteq \operatorname{dom} \delta$, we define the preliminary semantics[1] of P with respect to δ, written $[P]_\delta$, which returns a value:

[1] This is the *preliminary* semantics. Another semantics which builds on this one will be given later.

$[X]_\delta = \{x : \delta(X) \bullet \langle x \rangle\}$
$[P \texttt{ + } Q]_\delta = [P]_\delta \cup [Q]_\delta$
$[P \texttt{ -> } Q]_\delta = \{xs : [P]_\delta;\ ys : [Q]_\delta \bullet xs \frown ys\}$

$\diamond$

Example 2.11

Continuing with the assignments of Example 2.7, (`Professor + Student`) `->` `Subject` is a relational type. Its preliminary semantics with respect to δ is:

$[(\texttt{Professor + Student}) \texttt{ -> } \texttt{Subject}]_\delta =$
$\{\langle bill, concurrency\rangle,$
$\langle ben, concurrency\rangle,$
$\langle jack, concurrency\rangle,$
$\langle john, concurrency\rangle,$
$\langle bill, topology\rangle,$
$\langle ben, topology\rangle,$
$\langle jack, topology\rangle,$
$\langle john, topology\rangle\}$

$\diamond$

Relational types can be formed with a mixed arity e.g. $X \texttt{ + } (Y \texttt{ -> } Z)$ is the union of a unary, and a binary, relational type. Because Alloy does not allow mixed arity types [Jac06, pp. 110], and because they would cause us complications later on, we define a restriction on relational types to ensure they are of fixed arity.

Definition 2.12 Fixed arity types

We first define a function *Arities* returning the set of arities present in a relational type:

$Arities(X) = \{1\}$
$Arities(P \texttt{ + } Q) = Arities(P) \cup Arities(Q)$
$Arities(P \texttt{ -> } Q) = \{t : Arities(P);\ u : Arities(Q) \bullet t + u\}$

We then define:

$FixedArityType = \{P : RelationalType \mid \#(Arities(P)) = 1\}$

Note that:

$\#(Arities(P \texttt{ + } Q)) = 1 \Rightarrow \#(Arities(P)) = \#(Arities(Q)) = 1$
$\#(Arities(P \texttt{ -> } Q)) = 1 \Rightarrow \#(Arities(P)) = \#(Arities(Q)) = 1$

i.e. The relational types from which fixed arity types are constructed are also fixed arity ones. This allows us to use structural induction over *FixedArityType*. For the remainder of the thesis the letters P, Q will stand for fixed arity types.

◇

Definition 2.13 Multiplicity Keywords
Elements of the following set are called multiplicity keywords:

$$\mathit{Multi} ::= \texttt{set} \mid \texttt{one} \mid \texttt{lone} \mid \texttt{some}$$

◇

Multiplicity keywords will be used to enrich the syntax of types so that they can represent not just relations, but more specific forms of relations such as functions, bijections etc. We will use the letters m, n to stand for multiplicity keywords.

Definition 2.14 Semantics of multiplicity keywords
The meaning of each multiplicity keyword is a predicate over sets. Given any set s, we define:

$$[\![\texttt{set}]\!](s) \Leftrightarrow \mathit{true}$$
$$[\![\texttt{one}]\!](s) \Leftrightarrow \#s = 1$$
$$[\![\texttt{lone}]\!](s) \Leftrightarrow \#s \leq 1$$
$$[\![\texttt{some}]\!](s) \Leftrightarrow \#s \geq 1$$

To remember the meaning of `lone`, the reader may find it helpful to think of this standing for 'less than or equal to one'. ◇

We are now ready to define the complete syntax of type expressions in the language.

Definition 2.15 Type expressions
We define *RelType* and *TypeExp*, the syntax of type expressions as follows:

$$\begin{array}{rcl} \mathit{RelType} ::= & & \mathit{FixedArityType} \\ & \mid & \mathit{RelType}\ \mathit{Multi}\texttt{->}\mathit{Multi}\ \mathit{RelType} \end{array}$$

$$\mathit{TypeExp} ::= \quad \mathit{Multi}\ \mathit{RelType}$$

We use the letters T, U to stand for elements of *RelType* and *TypeExp*. ◇

The semantics of these types will be explained very shortly. We first extend the functions *Used* and *Arities* and the preliminary semantics to cover these types.

Definition 2.16 Used type variables, arities and preliminary semantics for type expressions
We extend the function *Used* (Definition 2.9) to cover type expressions as follows:

$$\mathit{Used}(T\ m\texttt{->}n\ U) = \mathit{Used}(T) \cup \mathit{Used}(U)$$
$$\mathit{Used}(m\ T) = \mathit{Used}(T)$$

We extend the function *Arities* to cover type expressions as follows:

$$Arities(T\ m\texttt{->}n\ U) = \{t : Arities(T);\ u : Arities(U) \bullet t + u\}$$
$$Arities(m\ T) = Arities(T)$$

We extend the preliminary semantics (Definition 2.10) to cover type expressions too. Given $T : TypeExp$ and $\delta : SetMap$ such that $Used(T) \subseteq \operatorname{dom}\delta$ we define:

$$[m\ T]_\delta = [T]_\delta$$
$$[T\ m\texttt{->}n\ U]_\delta = \{xs : [T]_\delta;\ ys : [U]_\delta \bullet xs \frown ys\}$$

◇

Note that in effect multiplicity keywords are ignored by the preliminary semantics. We shall shortly give a semantics which takes into account the multiplicity keywords.

The following lemma states that type expressions have a fixed arity.

Lemma 2.17 Types have a fixed arity
Let $T : TypeExp$. Then $\#(Arities(T)) = 1$. ◇

Proof The proof is by induction over the structure of T. The base case follows immediately from Definition 2.12. □

There is a large volume of proof in this thesis. So that the thesis does not become too long and tedious, we only sketch the proof for the relatively trivial lemmas such as the above.

We now give a semantics to type expressions which takes into account multiplicity keywords. The semantics is a set of values. Later on we will define the *variables* of the language and assign to each one a type expression. The semantics of the type expression tells us the possible values a variable may take.

Definition 2.18 Type semantics
Given $T : TypeExp$ and $\delta : SetMap$ such that $Used(T) \subseteq \operatorname{dom}\delta$, we define the semantics of T with respect to δ, written $[\![T]\!]_\delta$, as follows:

$$[\![P]\!]_\delta = \mathbb{P}([P]_\delta)$$

$$[\![m\ T]\!]_\delta = \{V : [\![T]\!]_\delta \mid [\![m]\!](V)\}$$

$$[\![T\ m\texttt{->}n\ U]\!]_\delta = \{rel : [T]_\delta \leftrightarrow [U]_\delta \mid$$
$$(\forall xs : [T]_\delta \bullet rel(\!|\ \{xs\}\ |\!) \in [\![n\ U]\!]_\delta) \land$$
$$(\forall ys : [U]_\delta \bullet rel^{\sim}(\!|\ \{ys\}\ |\!) \in [\![m\ T]\!]_\delta) \bullet$$
$$\{pair : rel \bullet first(pair) \frown second(pair)\}\}$$

N.B. $rel^{\sim}$ denotes the relational inverse of rel. ◇

The final clause above requires some explanation. A variable declared of the type T m`->`n U denotes a relation whose tuples are the concatenation of tuples in T and tuples in U. The keywords m and n constrain the relation. For each tuple $t1$ in T, the number of tuples in the relation beginning with $t1$ must satisfy the constraint given by n. For each tuple $t2$ in U, the number of tuples in the relation ending with $t2$ must satisfy the constraint given by m.

When T and U are unary, these reduce to familiar notions. For example, X `set->one` Y denotes total functions from X to Y. X `set->lone` Y denotes partial functions, and X `one->one` Y, bijections.

The type system in this language is based on Alloy's [Jac06, B.7.3], and is chosen to allow easy translation of formulas from Alloy.[2]

Example 2.19

Continuing with the definitions given in Example 2.7, the following is a type expression:

```
one Student
```

Its semantics with respect to δ is:

$$\{ \{\langle jack \rangle\}, \{\langle john \rangle\} \}$$

The following is a also type expression:

```
set (Professor one->one Subject)
```

It denotes the bijections between `Professor` and `Subject`. There are two possible bijections and so its semantics with respect to δ is:

$$\{ \{\langle bill, concurrency \rangle, \langle ben, topology \rangle\}, \{\langle bill, topology \rangle, \langle ben, concurrency \}\}$$

$\diamond$

The following lemma states that the function *Arities* tells us the length of the sequences of atoms in the values of a type.

Lemma 2.20

Let T : *TypeExp* and δ : *SetMap* such that $Used(T) \subseteq \operatorname{dom} \delta$. Let $V : [\![T]\!]_\delta$

[2] In Alloy, the multiplicity keyword in a type expression of the form *Multi RelType* has to be `set` unless *RelType* is unary, but we choose not to make this restriction. Later in the thesis we will cut down the range of possible type expressions, but in the early stages working with a more general range of possibilities actually makes proofs shorter.

and $xs : V$. Then:

$$\#(xs) \in Arities(T)$$

◇

Proof The proof is an simple induction over the structure of T. □

Convention 2.21 Default multiplicity keywords.

In a type expression T m`->`n U the multiplicity keywords m and n may be omitted. Their default value is `set`. Note that P `set->set` Q and P `->` Q have the same semantics so this does not introduce an ambiguity.

In a type expression m T the multiplicity marking m may be omitted. If T belongs to *FixedArityType* and $Arities(T) = \{1\}$ then T is unary and the default value is `one`. Otherwise the default value is `set`

This convention is used in the full-blown Alloy language, and we make use of it in Chapter 9. To reduce complexity we will otherwise avoid its use. ◇

2.2 Variables

This section introduces the *variables* of the language and explains how they are assigned values through the use of *bindings*. We will use variables in formulas, where they may stand for a range of values.

Definition 2.22 Variables

Let *Var* denote a set, whose elements we shall call variables. We denote an element of this set by *var*.

◇

Type maps are used to declare the type of each variable used in a modelling situation. The type given constrains the range of possible values to which a variable may be bound.

Definition 2.23 Type maps

A type map is a partial map from variables to type expressions.

$$TypeMap = Var \nrightarrow TypeExp$$

We will use the symbol Γ to denote a type map. ◇

Example 2.24

We use the assignments of Example 2.7 and let $Var = \{\texttt{studies}, \texttt{passes}\}$. The following is then a type map:

$\Gamma = \{$
  `studies` $\mapsto$ `set (Student set->one Subject)`
  `passes` $\mapsto$ `set Student`
$\}$

This tell us in the situation being modelled, `studies` is a total map from `Student` to `Subject` (for example representing the subject each student studies) and `passes` is a subset of `Student` (for example representing the set of students who pass their exams). ◇

Definition 2.25 Schemas

A schema is a pair consisting of a set of type variables together with a type map. The set of type variables contains the only ones which may be used by the type expressions in the type map.

$$\begin{aligned} \mathit{Schema} = \{\Upsilon : \mathbb{P}\, \mathit{TypeVar};\ \Gamma : \mathit{TypeMap} \mid \\ \forall\, \mathit{var} : \operatorname{dom}\Gamma \bullet \mathit{Used}(\Gamma(\mathit{var})) \subseteq \Upsilon\} \end{aligned}$$

We will in general denote a schema by (Υ, Γ). ◇

Schemas tell us the type variables and variables appropriate to a modelling situation. The type of each variable is specified by the type map. It is not always possible to infer the set of relevant type variables from the type map alone because the formulas of the language may contain, for example, quantification constructs as will be seen later on in Section 2.3. Therefore the set of relevant type variables is specified as well as the type map in a schema.

Recall that set maps are partial functions which apply to type variables. A set map is *appropriate* for a modelling situation if it is defined on all type variables relevant to that situation.

Definition 2.26 Appropriate set maps

Given a set of relevant type variables Υ, we define the set of appropriate set maps, denoted: $\mathit{SetMap}(\Upsilon)$, by:

$$\mathit{SetMap}(\Upsilon) = \{\delta : \mathit{SetMap} \mid \Upsilon \subseteq \operatorname{dom}\delta\}$$

◇

Definition 2.27 Bindings

In the context of a set of atoms, a schema (Υ, Γ) and an appropriate set map $\delta : \mathit{SetMap}(\Upsilon)$, a binding is a partial map from variables to values, which maps each variable in the domain of the type map, Γ, to a value belonging to the meaning of the type expression assigned to it by Γ:

$$\begin{aligned} \mathit{Binding}(\Upsilon, \Gamma, \delta) = \{\eta : \mathit{Var} \pfun \mathit{Value} \mid \\ \operatorname{dom}\eta = \operatorname{dom}\Gamma \wedge (\forall\, \mathit{var} : \operatorname{dom}\eta \bullet \eta(\mathit{var}) \in [\![\Gamma(\mathit{var})]\!]_\delta)\} \end{aligned}$$

We refer to an element of $\mathit{Binding}(\Upsilon, \Gamma, \delta)$ with the symbol η. ◇

Bindings are used to instantiate the variables in a formula with values.

Example 2.28

We reuse the assignments of Example 2.24 and Example 2.7. Let $\Upsilon = \{\texttt{Student}, \texttt{Subject}\}$. Then (Υ, Γ) is a schema. It follows $\delta \in \mathit{SetMap}(\Upsilon)$, and

the following is a binding:

$$
\begin{array}{l}
\{ \\
\quad \texttt{studies} \mapsto \{\langle jack, concurrency\rangle, \langle john, concurrency\rangle\}, \\
\quad \texttt{passes} \mapsto \{\langle jack\rangle\} \\
\}
\end{array}
$$

$\diamond$

Example 2.29

We have so far chosen a set of atoms to make the examples easier to understand, but in this example we emphasise that the atoms are uninterpreted. Let:

$$
\begin{array}{l}
Atom = \{a_1, a_2, a_3, a_4\} \\
TypeVar = \{\texttt{Professor}, \texttt{Student}, \texttt{Subject}, \texttt{City}\}
\end{array}
$$

and:

$$
\begin{array}{l}
\delta = \{ \\
\quad \texttt{Professor} \mapsto \{a_1, a_2\}, \\
\quad \texttt{Student} \mapsto \{a_3\}, \\
\quad \texttt{Subject} \mapsto \{a_4\} \\
\}
\end{array}
$$

Of course δ is still a set map. If we are interested in modelling a particular subject, say concurrency, then we would define a type map declaring a variable `concurrency` as follows:

$$
\begin{array}{l}
\Gamma = \{ \\
\quad \texttt{concurrency} \mapsto (\texttt{one Subject}) \\
\}
\end{array}
$$

$\diamond$

2.3 Formulas

In this section we define the syntax and semantics of the formulas and expressions of the language.

Definition 2.30 Syntax of expressions and formulas

The unary operators for expressions are transpose and transitive closure:

$$UnOp ::= \tilde{} \mid \hat{}$$

The binary operators for expressions are union, difference, intersection, join and

Cartesian product:

$$\mathit{BinOp} ::= \texttt{+} \mid \texttt{-} \mid \texttt{\&} \mid \texttt{.} \mid \texttt{->}$$

An expression can be a variable, a type variable, a unary operator applied to an expression, a binary operator applied to two expressions, or a set comprehension:

$$\begin{array}{rcl} \mathit{Expr} ::= & & \mathit{Var} \\ & \mid & \mathit{TypeVar} \\ & \mid & \mathit{UnOp}\ \mathit{Expr} \\ & \mid & \mathit{Expr}\ \mathit{BinOp}\ \mathit{Expr} \\ & \mid & \{\, \mathit{Var} : \mathit{TypeVar} \mid \mathit{Formula} \,\} \end{array}$$

The comparison operators are equality and inclusion:

$$\mathit{CompOp} ::= \texttt{=} \mid \texttt{in}$$

The binary logic operators for formulas are conjunction, disjunction, and implication.

$$\mathit{LogicOp} ::= \texttt{and} \mid \texttt{or} \mid \texttt{=>}$$

Quantification is either universal or existential:

$$\mathit{QuantOp} ::= \texttt{all} \mid \texttt{some}$$

A formula is either a comparison between two expressions, a binary logic operator applied to two formulas, the negation of a formula, or a quantified formula.

$$\begin{array}{rcl} \mathit{Formula} ::= & & \mathit{Expr}\ \mathit{CompOp}\ \mathit{Expr} \\ & \mid & \mathit{Formula}\ \mathit{LogicOp}\ \mathit{Formula} \\ & \mid & \texttt{!}\,\mathit{Formula} \\ & \mid & \mathit{QuantOp}\ \ \mathit{Var} : \mathit{TypeVar} \mid \mathit{Formula} \end{array}$$

$\diamond$

We will use the letters e, f to stand for expressions. We will use the letters F, G to stand for formulas.

We give a type system to our language. The type system ensures that the value of expressions has a fixed arity.[3]

Definition 2.31 Type system

We write $\Gamma \vdash e : i$ if the expression e is judged to have the type i in the context of Γ, where i is a natural number. Informally, i corresponds to the arity of the expression e.

[3] A stronger type system, more akin to the one used in Alloy, could be given to our language. However we do not need this stronger type system for the small model theorem and it would add complication to the thesis. There is no loss if our small model theorem applies to a wider range of formulas than those that would be permitted by the stronger type system. More information on Alloy's type system can be found in [EJT04].

We write $\Gamma \vdash F$ if the formula F is judged to be well-typed in the context of Γ. We also write $\Gamma \vdash e$ as shorthand for $\exists\, i : \mathbb{N} \bullet \Gamma \vdash e : i$.

Note that the judgment takes place in the context of Υ also, a fixed subset of *TypeVar* such that (Υ, Γ) is a schema. When $\Gamma \vdash F$ in the context of Υ we say that the schema (Υ, Γ) and F are *compatible*. Type judgements are defined using the inference system below:

$$\frac{}{\Gamma \vdash var : i} \quad [\, (var \mapsto T) \in \Gamma \wedge Arities(T) = \{i\} \,]$$

$$\frac{}{\Gamma \vdash X : 1} \quad [\, X \in \Upsilon \,]$$

$$\frac{\Gamma \vdash e : i}{\Gamma \vdash \sim e : i}$$

$$\frac{\Gamma \vdash e : 2}{\Gamma \vdash \hat{} \; e : 2}$$

$$\frac{\Gamma \vdash e : i \wedge \Gamma \vdash f : i}{\Gamma \vdash e \;\texttt{+}\; f : i}$$

$$\frac{\Gamma \vdash e : i \wedge \Gamma \vdash f : i}{\Gamma \vdash e \;\texttt{\&}\; f : i}$$

$$\frac{\Gamma \vdash e : i \wedge \Gamma \vdash f : i}{\Gamma \vdash e \;\texttt{-}\; f : i}$$

$$\frac{\Gamma \vdash e : i \wedge \Gamma \vdash f : j}{\Gamma \vdash e \;\texttt{.}\; f : i + j - 2} \quad [\, i > 0 \wedge j > 0 \,]$$

$$\frac{\Gamma \vdash e : i \wedge \Gamma \vdash f : j}{\Gamma \vdash e \;\texttt{->}\; f : i + j}$$

$$\frac{\Gamma \vdash e : i \wedge \Gamma \vdash f : i}{\Gamma \vdash e \;\texttt{in}\; f}$$

$$\frac{\Gamma \vdash e : i \land \Gamma \vdash f : i}{\Gamma \vdash e \texttt{ = } f}$$

$$\frac{\Gamma \vdash F \land \Gamma \vdash G}{\Gamma \vdash F \texttt{ and } G}$$

$$\frac{\Gamma \vdash F \land \Gamma \vdash G}{\Gamma \vdash F \texttt{ or } G}$$

$$\frac{\Gamma \vdash F \land \Gamma \vdash G}{\Gamma \vdash F \texttt{ => } G}$$

$$\frac{\Gamma \vdash F}{\Gamma \vdash \texttt{!}F}$$

$$\frac{\Gamma \oplus \{var \mapsto (\texttt{one } X)\} \vdash F}{\Gamma \vdash \texttt{all } var\texttt{:}X\texttt{ | }F} \quad [\, X \in \Upsilon \,]$$

$$\frac{\Gamma \oplus \{var \mapsto (\texttt{one } X)\} \vdash F}{\Gamma \vdash \texttt{some } var\texttt{:}X\texttt{ | }F} \quad [\, X \in \Upsilon \,]$$

$\diamond$

These following functions will allow us to reuse Z's transitive closure operator in defining the semantics of transitive closure in our language.

Definition 2.32 Conversion between sets of lists and sets of pairs

We define functions to convert between sets of lists (of size 2) and sets of pairs:

$$Lists2Pairs(V) = \{xs : V \mid \#(xs) = 2 \bullet (xs(1), xs(2))\}$$

$$Pairs2Lists(pairs) = \{pair : pairs \bullet \langle first(pair), second(pair)\rangle\}$$

$\diamond$

We now give a semantics to formulas and expressions. Given a binding for the relevant variables and a set map for the relevant type variables, the semantics of an expression assigns it a value, and the semantics of a formula assigns it either *true* or *false* .

Definition 2.33 Semantics of expressions and formulas

Let (Υ, Γ) be a schema. Let F be a formula (resp. e, an expression), such that $\Gamma \vdash F$ (resp. $\Gamma \vdash e$) . Let $\delta : SetMap(\Upsilon)$ and $\eta : Binding(\Upsilon : \Gamma, \delta$. Below, we define the semantics of F (resp. semantics of e) with respect to δ and η written $[\![F]\!]^\eta_\delta$ (resp. $[\![e]\!]^\eta_\delta$).

$$[\![var]\!]^\eta_\delta = \eta(var)$$

$$[\![X]\!]^\eta_\delta = \{x : \delta(X) \bullet \langle x \rangle\}$$

$$[\![\texttt{~}\ e]\!]^\eta_\delta = \{s : [\![e]\!]^\eta_\delta \bullet rev(s)\}$$

$$[\![\texttt{\^{}}\ e]\!]^\eta_\delta = Pairs2Lists((Lists2Pairs([\![e]\!]^\eta_\delta))^+)$$

$$[\![e\ \texttt{+}\ f]\!]^\eta_\delta = [\![e]\!]^\eta_\delta \cup [\![f]\!]^\eta_\delta$$

$$[\![e\ \texttt{-}\ f]\!]^\eta_\delta = [\![e]\!]^\eta_\delta \setminus [\![f]\!]^\eta_\delta$$

$$[\![e\ \texttt{\&}\ f]\!]^\eta_\delta = [\![e]\!]^\eta_\delta \cap [\![f]\!]^\eta_\delta$$

$$[\![e\ \texttt{.}\ f]\!]^\eta_\delta = \{s : [\![e]\!]^\eta_\delta;\ t : [\![f]\!]^\eta_\delta \mid last(s) = head(t) \bullet front(s) \frown tail(t)\}$$

$$[\![e\ \texttt{->}\ f]\!]^\eta_\delta = \{s : [\![e]\!]^\eta_\delta;\ t : [\![f]\!]^\eta_\delta \bullet s \frown t\}$$

$$[\![\{var : X \mid F\}]\!]^\eta_\delta = \{x : \delta(X) \mid [\![F]\!]^{\eta \oplus \{var \mapsto \{\langle x \rangle\}\}}_\delta \bullet \langle x \rangle\}$$

$$[\![e\ \texttt{in}\ f]\!]^\eta_\delta \Leftrightarrow [\![e]\!]^\eta_\delta \subseteq [\![f]\!]^\eta_\delta$$

$$[\![e\ \texttt{=}\ f]\!]^\eta_\delta \Leftrightarrow [\![e]\!]^\eta_\delta = [\![f]\!]^\eta_\delta$$

$$[\![F\ \texttt{and}\ G]\!]^\eta_\delta \Leftrightarrow [\![F]\!]^\eta_\delta \wedge [\![G]\!]^\eta_\delta$$

$$[\![F\ \texttt{or}\ G]\!]^\eta_\delta \Leftrightarrow [\![F]\!]^\eta_\delta \vee [\![G]\!]^\eta_\delta$$

$$[\![F\ \texttt{=>}\ G]\!]^\eta_\delta \Leftrightarrow [\![F]\!]^\eta_\delta \Rightarrow [\![G]\!]^\eta_\delta$$

$$[\![\,!\, F]\!]^{\eta}_{\delta} \Leftrightarrow \neg\, [\![F]\!]^{\eta}_{\delta}$$

$$[\![\,\texttt{all}\ \ var : X \mid F]\!]^{\eta}_{\delta} \Leftrightarrow (\forall\, x : \delta(X) \bullet [\![F]\!]^{\eta \oplus \{var \mapsto \{\langle x \rangle\}\}}_{\delta})$$

$$[\![\,\texttt{some}\ \ var : X \mid F]\!]^{\eta}_{\delta} \Leftrightarrow (\exists\, x : \delta(X) \bullet [\![F]\!]^{\eta \oplus \{var \mapsto \{\langle x \rangle\}\}}_{\delta})$$

◇

The following lemma confirms that the type of an expression is indeed its arity, i.e. the length of the sequences of the value assigned to it by the semantics.

Lemma 2.34
Let (Υ, Γ) be a schema. Let $e : \mathit{Expr}$ and $i : \mathbb{N}$. Suppose $\Gamma \vdash e : i$. Let $\delta : \mathit{SetMap}(\Gamma)$ and $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$. Then:

$$\forall\, xs : [\![e]\!]^{\eta}_{\delta} \bullet \#(xs) = i$$

◇

Proof The proof is an induction over the structure of e, with the statement of the lemma as the inductive hypothesis. Lemma 2.20 gives the base case of the induction. □

Example 2.35
We define a schema (Υ, Γ) by:

$$\begin{aligned} \Upsilon &= \{\texttt{Student}, \texttt{Subject}\} \\ \Gamma &= \{\texttt{studies} \mapsto\ \ \texttt{set}\ (\texttt{Student}\ \texttt{set->one}\ \texttt{Subject}) \\ &\qquad \texttt{passes} \mapsto\ \ \texttt{set}\ \texttt{Student}\} \end{aligned}$$

The following formula is compatible:

```
some sub:Subject|(!((studies . sub) in passes))
```

and could be translated into natural language as: there is a subject and not all students who study this subject pass their exam.

Here is one possible set map, δ, and binding, η, for the formula:

$$\begin{aligned} \delta = \{\texttt{Student} \mapsto\ & \{\mathit{jack}, \mathit{john}\}, \\ \texttt{Subject} \mapsto\ & \{\mathit{concurrency}, \mathit{topology}\}\} \\ \eta = \{\texttt{studies} \mapsto\ & \{\langle \mathit{jack}, \mathit{concurrency} \rangle, \\ & \langle \mathit{john}, \mathit{topology} \rangle\}, \\ \texttt{passes} \mapsto\ & \{\langle \mathit{jack} \rangle\}\} \end{aligned}$$

which could be translated into natural language as: the students are Jack and John and the subjects are Concurrency and Topology. Jack studies Concurrency and John studies Topology. Only Jack passes his exam. Using Definition 2.33

we can determine whether the formula is true with this binding:

$$
\begin{array}{l}
[\![(\texttt{studies . sub})]\!]_{\delta}^{\eta \oplus \{sub \mapsto \{\langle topology \rangle\}\}} \\
= \\
\{s : \{\langle jack, concurreny \rangle, \langle john, topology \rangle\}; \\
\quad t : \{\langle topology \rangle\} \mid last(s) = head(t) \bullet front(s) \frown tail(t)\} \\
= \\
\{\langle john \rangle\}
\end{array}
$$

Hence:

$$
\begin{array}{l}
[\![(\texttt{(studies . sub) in passes})]\!]_{\delta}^{\eta \oplus \{sub \mapsto \{\langle topology \rangle\}\}} \\
\Leftrightarrow \\
\{\langle john \rangle\} \subseteq \{\langle jack \rangle\} \\
\Leftrightarrow \\
false
\end{array}
$$

Hence:

$$
\begin{array}{l}
[\![(\texttt{!((studies . sub) in passes))})]\!]_{\delta}^{\eta \oplus \{sub \mapsto \{\langle topology \rangle\}\}} \\
\Leftrightarrow \\
true
\end{array}
$$

Hence:

$$
\begin{array}{l}
[\![\ \texttt{some sub:Subject|(!((studies . sub) in passes))})]\!]_{\delta}^{\eta} \\
\Leftrightarrow \\
\exists\, x : \delta(Subject) \bullet [\![(\texttt{!((studies . sub) in passes))})]\!]_{\delta}^{\eta \oplus \{sub \mapsto \{\langle x \rangle\}\}} \\
\Leftrightarrow \\
true
\end{array}
$$

Hence the formula is true under this binding.

However, under this binding:

$$
\begin{array}{rll}
\delta = \{\texttt{Student} \mapsto & \{jack, john\}, \\
\texttt{Subject} \mapsto & \{concurrency, topology\}\} \\
\eta = \{\texttt{studies} \mapsto & \{\langle jack, concurrency \rangle, \\
 & \langle john, topology \rangle\}, \\
\texttt{passes} \mapsto & \{\langle jack \rangle, \\
 & \langle john \rangle\}\}
\end{array}
$$

which is the same, except now John passes his exams, the formula is false. $\diamond$

Definition 2.36 Consistent and valid formulas

Given a formula F and a compatible schema (Υ, Γ) we say 'F is consistent' if there exists an appropriate set map and binding for which the semantic function

on F is true.

$$\begin{array}{l} \mathit{Consistent}(F, \Upsilon, \Gamma) \Leftrightarrow \\ (\exists\, \delta : \mathit{SetMap}(\Upsilon) \bullet \exists\, \eta : \mathit{Binding}(\Upsilon, \Gamma, \delta) \bullet [\![F]\!]^{\eta}_{\delta}) \end{array}$$

We say 'F is valid' if the negation of F is not consistent.

$$\mathit{Valid}(F, \Upsilon, \Gamma) \Leftrightarrow \neg\ \mathit{Consistent}(!\, F, \Upsilon, \Gamma)$$

$\diamond$

2.4 Scopes

The Alloy language, being an extension of first order logic, is not decidable. It is not possible for the Alloy Analyzer program to determine whether a formula is valid. However, the Alloy Analyzer can determine whether a formula is valid within a finite *scope*. The finite scope places a bound on the size of every carrier set.

Definition 2.37 Scopes

We first define a type $\mathbb{N}^{\infty}$ to consist of the naturals together with ∞. We extend the arithmetic operations of sum and product to cover $\mathbb{N}^{\infty}$ in the usual way: both the sum and product of ∞ together with any other value yields ∞. We extend the comparison operator $\leq$ in the usual way: $\forall\, i : \mathbb{N}^{\infty} \bullet i \leq \infty$. We also extend the cardinality operator $\#$ to return ∞ on infinite sets.

Given a set of type variables Υ, a scope for Υ, denoted: $\mathit{Scope}(\Upsilon)$ is a total map from Υ to $\mathbb{N}^{\infty}$:

$$\mathit{Scope}(\Upsilon) = \Upsilon \rightarrow \mathbb{N}^{\infty}$$

$\diamond$

We will use the symbol Θ for a scope.

Definition 2.38 Finite and infinite scopes

A scope is infinite if it contains ∞ in its range, otherwise it is finite.

$$\mathit{Infinite}(\Theta) \Leftrightarrow \infty \in \operatorname{ran} \Theta$$

$$\mathit{Finite}(\Theta) \Leftrightarrow \infty \notin \operatorname{ran} \Theta$$

$\diamond$

Definition 2.39 Ordering of scopes

Scopes can be partially ordered using the point-wise ordering. Given a set of type variables Υ and given $\Theta_1 : \mathit{Scope}(\Upsilon)$ and $\Theta_2 : \mathit{Scope}(\Upsilon)$ we define $\Theta_1 \leq \Theta_2$ as follows:

$$\Theta_1 \leq \Theta_2 \Leftrightarrow (\forall X : \Upsilon \bullet \Theta_1(X) \leq \Theta_2(X))$$

$\diamond$

Definition 2.40 Consistent and valid formulas within a scope

Given a formula F and a compatible schema (Υ, Γ) and a scope $\Theta : \mathit{Scope}(\Upsilon)$, we say '$F$ is consistent within Θ' if there exists a compatible set map which assigns to each type variable X a carrier set no larger than $\Theta(X)$ and a binding for which the semantic function on F is true.

$$(F, \Upsilon, \Gamma)\ \mathit{ConsistentWithin}\ \Theta \Leftrightarrow$$
$$(\exists\, \delta : \mathit{SetMap} \mid \mathrm{dom}\, \delta = \Upsilon \wedge (\# \circ \delta) \leq \Theta \bullet \exists\, \eta : \mathit{Binding}(\Upsilon, \Gamma, \delta) \bullet [\![F]\!]^{\eta}_{\delta})$$

We say 'F is valid within Θ' if the negation of F is not consistent within Θ.

$$(F, \Upsilon, \Gamma)\ \mathit{ValidWithin}\ \Theta \Leftrightarrow \neg\ (!\, F, \Upsilon, \Gamma)\ \mathit{ConsistentWithin}\ \Theta$$

$\diamond$

The Alloy Analyzer can determine whether a formula is valid/consistent within a finite scope i.e. this problem is decidable. For the purposes of looking for a binding for a formula which makes it true of false, it follows due to the uninterpreted nature of atoms that, two set maps can be considered isomorphic if they agree on the size of carrier set of every type variable.

Furthermore, there are only a finite number of possible bindings for a fixed schema and a set map which assigns to each type variable a finite carrier set. Consequently the Alloy Analyzer just needs to check a finite number of possibilities to see if a formula is valid/consistent within a finite scope. (In practice the number is very large and the Alloy Analyzer reduces this number using symmetry [JJD98]).

Summary

In this chapter we defined the language we use in the thesis, based on the Alloy Kernel language. The language has an unusual type system in that the type of every variable is a relation, but we can represent subsets as unary relations and scalars as singleton unary relations. The type system is enriched with multiplicity keywords which allow not just simple relations to be defined, but also more specific forms of relations such as function, bijections etc.

We gave a semantics for the expressions and formulas of the language and explained what it means for a formula to be consistent or valid. We formally defined the notion of scope which is used by the Alloy Analyzer and explained what it means for a formula to be consistent or valid within a scope.

Our language differs from the Alloy Kernel language only by keeping the syntax of types from the full-blown language. We keep the types from the full-blown language because translation to the type system of the Alloy Kernel language would lose information that can be used to keep thresholds smaller

than they would otherwise be. The next chapter will define a notion of semantic data independence, which will be our first step towards using the Alloy Analyzer to check formulas at infinite scopes.

Chapter 3

Semantic data independence

In this chapter we give a semantic definition of data independence. Formulas that satisfy this definition have a property which allows us to evaluate their truth under some binding by instead using a related *quotient* binding.

The small model theorem uses this property. When checking such a formula, one can replace all the relevant bindings with appropriate quotient bindings. The advantage is that one can often choose quotient bindings which lie within a small scope. How one makes these choices and exactly how small a scope can be used will be investigated in Chapter 5.

3.1 Quotient bindings

In this section we define *quotient bindings*. Quotient bindings can be considered as approximations to bindings which lie within a smaller scope.

We start by defining *quotient atoms*, which are used by quotient bindings instead of atoms. Quotient atoms are defined by an equivalence relation acting on the set of atoms.

Definition 3.1 Equivalent atoms

We will use the symbol $\sim$ to denote an equivalence relation on $Atom$, and denote all such relations $EqRel$. Given a particular atom x, $[x]_\sim$ shall denote its equivalence class. $\diamond$

Definition 3.2 Quotient atoms and values

In the context of an equivalence relation on atoms, $\sim$, we define the set of quotient atoms as follows:

$$Atom^\sim = \{x : Atom \bullet [x]_\sim\}$$

This gives rise to an alternative set of values called quotient values:

$$Value^{\sim} = \mathbb{P}(\operatorname{seq} Atom^{\sim})$$

We denote a quotient atom by x' or y' and a sequence of quotient atoms, xs' or ys'. V' and W' will denote a quotient value. $\diamond$

Recall that we introduced atoms as a basis for the semantic values of expressions in the language. We can also use quotient atoms for this purpose and it makes no difference to the language semantics which is used. However, quotient atoms contain extra structure which allow us to define a semantic notion of data independence. A quotient atom can be considered an approximation to the set of atoms it contains.

We refer to the set maps and bindings that arise when using quotient atoms as *quotient* set maps and *quotient* bindings and define the formally as follows.

Definition 3.3 Quotient set maps

A quotient set map is just like a set map (see Definition 2.6), but maps type variables to sets of quotient atoms rather than sets of atoms.

$$\begin{aligned} &SetMap^{\sim} = \{\delta' : TypeVar \pfun \mathbb{P}\, Atom^{\sim} \\ &\quad \mid \forall X, Y : \operatorname{dom} \delta' \\ &\quad \bullet\ X \neq Y \Rightarrow \delta(X) \cap \delta(Y) = \emptyset\} \end{aligned}$$

We will use the symbol δ' to denote a member of $SetMap^{\sim}$. $\diamond$

Definition 3.4 Quotient bindings

A quotient binding, is just like a binding (see Definition 2.27), but assigns quotient values instead of values to variables. Given a schema (Υ, Γ) and a quotient set map δ' such that $\Upsilon \subseteq \operatorname{dom} \delta'$ we define the set of quotient bindings:

$$\begin{aligned} &Binding^{\sim}(\Upsilon, \Gamma, \delta') = \{\eta' : Var \pfun Value^{\sim} \mid \\ &\quad \operatorname{dom} \eta' = \operatorname{dom} \Gamma \land (\forall var : \operatorname{dom} \eta' \bullet \eta'(var) \in [\![\Gamma(var)]\!]_{\delta'})\} \end{aligned}$$

$\diamond$

3.2 Galois insertion

In this section we define a Galois insertion [DP02, chapter 7] between values with the subset order and quotient values with the subset order.

We first define the general notion of a Galois insertion.

Definition 3.5 Partial order

A partial order is a pair $(\alpha, \leq)$ where α is a set and $\leq$ is a binary relation on α which is reflexive, antisymmetric, and transitive. That is for all a, b, c in

α:

$$a \leq a$$
$$a \leq b \wedge b \leq a \Rightarrow a = b$$
$$a \leq b \wedge b \leq c \Rightarrow a \leq c$$

$\diamond$

Definition 3.6 Galois connection and insertion

Let $(\alpha, \leq)$ and $(\gamma, \sqsubseteq)$ be two partial orders. Let $H : \alpha \to \gamma$ and $I : \gamma \to \alpha$ be two functions. H and I are said to form a Galois connection between the two partial orders exactly when for any $a : \alpha$ and $b : \gamma$:

$$H(a) \sqsubseteq b \Leftrightarrow a \leq I(b)$$

H is called the lower adjoint of the connection and I, the upper adjoint.

If in addition $H \circ I$ is the identity function on γ, then H and I are said to form a Galois insertion. $\diamond$

The existence of a Galois connection between $(\alpha, \leq)$ and $(\gamma, \sqsubseteq)$, allows one to represent elements of α with elements of β which may be simpler or more abstract. In general there may be several elements of β which represent a particular element of α, and this means that β may contain elements which are not relevant for the approximation of α. The concept of Galois insertion rectifies this, because it implies that upper adjoint is injective.

Lemma 3.7 The upper adjoint of a Galois insertion is injective.

Let $(\alpha, \leq)$ and $(\gamma, \sqsubseteq)$ be two partial orders. Let $H : \alpha \to \gamma$ and $I : \gamma \to \alpha$ be a Galois insertion between the partial orders.

To show I is injective, we let $b, c : \gamma$ and suppose:

$$I(b) = I(c)$$

To complete the proof we show $b = c$ as follows:

$$I(b) = I(c)$$
$$\Rightarrow$$
$$I(b) \leq I(c) \wedge I(c) \leq I(b)$$
$$\Rightarrow$$
$$H(I(b)) \sqsubseteq c \wedge H(I(c)) \sqsubseteq b$$
$$\Rightarrow$$
$$b \sqsubseteq c \wedge c \sqsubseteq b$$
$$\Rightarrow$$
$$b = c$$

$\diamond$

To define a semantic notion of data independence we want to know when the approximation of a binding by a quotient binding is good enough. i.e. using the

approximation gives the same result in evaluating the truth of a formula. By using a Galois insertion we can compare values with quotient values and hence compare the interpretation of expressions and formulas in the context of atoms and quotient atoms.

We first define the *quotient of* a value. This turns out to be the lower adjoint of the Galois insertion.

Definition 3.8 Quotient of a value

Given an equivalence relation on atoms, $\sim$, and a sequence of atoms ys, we define the quotient of ys, written $ys/\sim$ by induction:

$$\langle\rangle/\sim = \langle\rangle$$
$$(\langle x\rangle \frown xs)/\sim = \langle [x]_\sim\rangle \frown (xs/\sim)$$

Note that the quotient operation: $/\sim$ distributes over concatenation.

We then define the quotient of a value V written $V/\sim$ by:

$$V/\sim = \{xs : V \bullet xs/\sim\}$$

Note that the quotient operation: $/\sim$ distributes over union, and is monotone with respect to the subset ordering. Note also that $V/\sim$ is a quotient value. $\diamond$

We now define the product of a quotient value. This turns out to be the upper adjoint of the Galois insertion.

Definition 3.9 Product of a quotient value

Given an equivalence relation on atoms, $\sim$, and a sequence of quotient atoms ys', we define the product of ys', written $ys'.\sim$ by induction:

$$\langle\rangle.\sim = \{\langle\rangle\}$$
$$(\langle x'\rangle \frown xs').\sim = \{x : x';\ xs : (xs'.\sim) \bullet \langle x\rangle \frown xs\}$$

We then define the product of a quotient value V' by:

$$V'.\sim = \bigcup\{xs' : V' \bullet xs'.\sim\}$$

Note that the product operation: $.\sim$ is monotone with respect to the subset ordering. Note also that $V'.\sim$ is a value. $\diamond$

Example 3.10

We give an example of the quotient of a value and the product of a quotient value. Let:

$$Atom = \{london, manchester, new\ york, philadelphia\}$$
$$\sim = \{x : Atom;\ y : Atom \mid \{x, y\} \subseteq \{london, manchester\}$$
$$\quad \lor \{x, y\} \subseteq \{new\ york, philadelphia\} \bullet (x, y)\}$$

Thus $\sim$ equates *london* with *manchester* and *new york* with *philadelphia*.

The quotient atoms are the equivalence classes:

$$Atom^{\sim} = \{\{london, manchester\}, \{new\ york, philadelphia\}\}$$

Let V be a value which is defined by:

$$V = \{\langle london\rangle, \langle manchester\rangle, \langle new\ york\rangle\}$$

It follows the quotient of V is:

$$V/\sim = \{\langle\{london, manchester\}\rangle, \langle\{new\ york, philadelphia\}\rangle\}$$

Let W' be a quotient value which is defined by:

$$W' = \{\langle\{london, manchester\}, \{new\ york, philadelphia\}\rangle\}$$

It follows the product of W' is:

$$\begin{array}{ll} W'.\sim = \{ & \langle london, new\ york\rangle, \\ & \langle london, philadelphia\rangle, \\ & \langle manchester, new\ york\rangle, \\ & \langle manchester, philadelphia\rangle\} \end{array}$$

◇

Before proving we have formed a Galois insertion, we prove some subsidiary lemmas.

Lemma 3.11
Let $ys : \mathrm{seq}\, Atom$ and $\sim : EqRel$. Then:

$$ys \in (ys/\sim).\sim$$

◇

Proof See Lemma A.9. □

Lemma 3.12
Let $\sim : EqRel$ and $xs' : \mathrm{seq}\, Atom^{\sim}$. Then:

$$(xs'.\sim)/\sim = \{xs'\}$$

◇

Proof See Lemma A.10. □

Lemma 3.13 Let $V : \mathrm{seq}\, Atom$. Let $\sim : EqRel$. Then:

$$V \subseteq (V/\sim).\sim$$

◇

Proof Let $xs : V$.

$$
\begin{array}{ll}
& xs \in V \\
\Rightarrow & [3.8] \\
& xs/\sim \in V/\sim \\
\Rightarrow & [3.11,\ 3.9] \\
& xs \in (xs/\sim).\sim \subseteq (V/\sim).\sim \\
\Rightarrow & \\
& xs \in (V/\sim).\sim
\end{array}
$$

□

Lemma 3.14 Let $\sim : \mathit{EqRel}$ and $W' : \mathit{Value}^{\sim}$. Then

$$W' = (W'.\sim)/\sim$$

◇

Proof We first show $(W'.\sim)/\sim \subseteq W'$. Let $xs' : (W'.\sim)/\sim$. Using Definition 3.8, one can choose $xs : W'.\sim$ such that $xs' = xs/\sim$. Then, by Definition 3.9 one can choose $ys' : W'$ such that $xs \in ys'.\sim$. It follows:

$$
\begin{array}{ll}
& xs \in ys'.\sim \\
\Rightarrow & [3.8] \\
& xs/\sim \in (ys'.\sim)/\sim \\
\Rightarrow & [3.12] \\
& xs' = xs/\sim \in (ys'.\sim)/\sim = \{ys'\} \\
\Rightarrow & \\
& xs' = ys' \in W'
\end{array}
$$

To complete the proof we show $W' \subseteq (W'.\sim)/\sim$. So now let $xs' : W'$.

$$
\begin{array}{ll}
& xs' \in W' \\
\Rightarrow & [3.9] \\
& xs'.\sim \subseteq W'.\sim \\
\Rightarrow & [3.8] \\
& (xs'.\sim)/\sim \subseteq (W'.\sim)/\sim \\
\Rightarrow & [3.12] \\
& \{xs'\} \subseteq (W'.\sim)/\sim \\
\Rightarrow & \\
& xs' \in (W'.\sim)/\sim
\end{array}
$$

□

Lemma 3.15 Galois Insertion between $Value$ and $Value^{\sim}$

Let $\sim : EqRel$. Note that $(Value, \subseteq)$ and $(Value^{\sim}, \subseteq)$ are partial orders.

Define $H : Value \to Value^{\sim}$ and $I : Value^{\sim} \to Value$ by:

$$H(V) = V/\sim$$
$$I(V') = V'.\sim$$

Then H and I form a Galois insertion between $(Value, \subseteq)$ and $(Value^{\sim}, \subseteq)$ where H is the lower adjoint and I is the upper adjoint. $\diamond$

Proof Let $V : Value$ and $W' : Value'$. We first show

$$H(V) \subseteq W' \Leftrightarrow V \subseteq I(W')$$

which will establish that H and I form a Galois connection (defined in Definition 3.6).

$$\begin{array}{ll}
H(V) \subseteq W' & \\
\Rightarrow & \\
V/\sim \subseteq W' & \\
\Rightarrow & [3.9] \\
(V/\sim).\sim \subseteq W'.\sim & \\
\Rightarrow & [3.13] \\
V \subseteq (V/\sim).\sim \subseteq W'.\sim & \\
\Rightarrow & \\
V \subseteq W'.\sim & \\
\Rightarrow & \\
V \subseteq I(W') &
\end{array}$$

$$\begin{array}{ll}
V \subseteq I(W') & \\
\Rightarrow & \\
V \subseteq W'.\sim & \\
\Rightarrow & [3.8,3.14] \\
V/\sim \subseteq (W'.\sim)/\sim = W' & \\
\Rightarrow & \\
V/\sim \subseteq W' & \\
\Rightarrow & \\
H(V) \subseteq W' &
\end{array}$$

That H and I form a Galois insertion (also defined in Definition 3.6) now follows immediately from Lemma 3.14.

$\square$

We define closed values in the usual way for Galois connections.

Definition 3.16 Closed values

In the context of an equivalence relation on atoms, $\sim$, a value V is said to be closed if the product of its quotient is itself.

$$\mathit{Closed}_{\sim}(V) \Leftrightarrow (V/{\sim}).{\sim} = V$$

$\diamond$

Closed values can be thought of as those values which have a good approximation by quotient value — good in the sense the the quotient value contains enough information to reconstruct the value.

Example 3.17

We give an example of a value which is closed and a value which is not.

Using the assignments in Example 3.10, it follows:

$$\begin{array}{l}(V/{\sim}).{\sim} \\ = \\ \{\langle\{\mathit{london}, \mathit{manchester}\}\rangle, \langle\{\mathit{new\ york}, \mathit{philadelphia}\}\rangle\}.{\sim} \\ = \\ \{\langle\mathit{london}\rangle, \langle\mathit{manchester}\rangle, \langle\mathit{new\ york}\rangle, \langle\mathit{philadelphia}\rangle\} \\ \neq \\ V\end{array}$$

So V is not closed. Let:

$$W = \{\langle\mathit{london}\rangle, \langle\mathit{manchester}\rangle\}$$

Then:

$$\begin{array}{l}(W/{\sim}).{\sim} \\ = \\ \{\langle\{\mathit{london}, \mathit{manchester}\}\rangle\}.{\sim} \\ = \\ W\end{array}$$

So W is a closed value. $\diamond$

3.3 The quotient of a binding

In this section we introduce the *quotient of* a binding. The quotient of a binding is the correct quotient binding to use when evaluating the truth of a formula which is semantically-data-independent i.e. the one which should give the same result as using the binding. In this sense it can be considered an approximation to the binding. Before we can define the quotient of a binding we introduce a number of other definitions.

Definition 3.18 Data relations

A data relation is a special type of equivalence relation on atoms. Given a set map δ, a data relation is a relation which does not equate atoms belonging to carrier sets of distinct type variables. The set of such relations is:

$$\begin{aligned} &\mathit{DataRel}(\delta) = \{\sim : \mathit{EqRel} \mid \forall\, X : \operatorname{dom}\delta;\ y : \mathit{Atom} \bullet \\ &\quad \forall\, x : \delta(X) \bullet x \sim y \Rightarrow y \in \delta(X)\} \end{aligned}$$

$\diamond$

The reason we consider only data relations rather than the more general equivalence relations on atom is that it is a feature of our language that carrier sets of distinct type variables are disjoint (see Definition 2.6). If we could equate atoms in the carrier sets of distinct type variables, there would be an issue as to the carrier set of which type variable the equivalence class (i.e. quotient atom) should lie, since both is not an option.

Next we introduce the *quotient of* a set map. Recall that a binding (Definition 2.27) exists in the context of a set map, and a quotient binding (Definition 3.4) exists in the context of a quotient set map. Therefore, before defining the quotient of a binding, we need to define the quotient set map in the context of which it exists. To define this we take the set map in the context of which the binding exists and form the quotient of this set map.

Definition 3.19 Quotient of a set map

Given any set map, δ, and any data relation, $\sim : \mathit{DataRel}(\delta)$, we define the quotient of δ by $\sim$ written $\delta/\sim$ by:

$$\delta/\sim = \{\mathit{maplet} : \delta \bullet (\mathit{first}(\mathit{maplet}), \{x : \mathit{second}(\mathit{maplet}) \bullet [x]_\sim\})\}$$

$\diamond$

We are now ready to define the quotient of a binding.

Definition 3.20 Quotient of a binding

Let (Υ, Γ) be a schema and $\delta : \mathit{SetMap}(\Upsilon)$. Let $\sim : \mathit{DataRel}(\delta)$ and $\eta : \mathit{Binding}(\Upsilon, \Gamma)$. We define the quotient of η by $\sim$, written $\eta/\sim$, by:

$$\begin{aligned} &\operatorname{dom}(\eta/\sim) = \operatorname{dom}\eta \\ &\forall\, \mathit{var} : \operatorname{dom}\eta \bullet (\eta/\sim)(\mathit{var}) = \eta(\mathit{var})/\sim \end{aligned}$$

$\diamond$

Example 3.21

We give an example of the quotient of a set map and the quotient of a binding. We reuse the assignments in Example 3.10 and additionally let:

$$\begin{aligned} &\Upsilon = \{\texttt{City}\} \\ &\Gamma = \{\texttt{usa} \mapsto \texttt{set}\ \mathit{City}\} \\ &\delta = \{\texttt{City} \mapsto \{\mathit{london}, \mathit{manchester}, \mathit{new\ york}, \mathit{philadelphia}\}\} \\ &\eta = \{\texttt{usa} \mapsto \{\langle \mathit{new\ york}\rangle, \langle \mathit{philadelphia}\rangle\}\} \end{aligned}$$

Thus (Υ, Γ) is a schema. $\delta \in \mathit{SetMap}(\Upsilon)$ and $\eta \in \mathit{Binding}(\Upsilon, \Gamma, \delta)$.

The quotient set map is:

$$\delta/\sim = \\ \{\mathtt{City} \mapsto \{\{\mathit{london}, \mathit{manchester}\}, \{\mathit{new\ york}, \mathit{philadelphia}\}\}$$

and $\delta/\sim \in \mathit{SetMap}^\sim$.

The quotient binding is:

$$\eta/\sim = \\ \{\mathtt{usa} \mapsto \{\langle\{\mathit{new\ york}, \mathit{philadelphia}\}\rangle\}$$

and $\eta/\sim \in \mathit{Binding}^\sim(\Upsilon, \Gamma, \delta/\sim)$. ◇

The rest of this section will be concerned with finding the sufficient conditions for the the quotient of a binding to be a quotient binding.

Lemma 3.22 The quotient of a set map by a data relation is a quotient set map.

Let δ be a set map and let $\sim : \mathit{DataRel}(\delta)$ be a data relation. Then:

$$\delta/\sim \in \mathit{TypeVar} \nrightarrow \mathbb{P}\, \mathit{Atom}^\sim$$

$$\mathrm{dom}(\delta/\sim) = \mathrm{dom}\, \delta$$

$$\delta/\sim \in \mathit{SetMap}^\sim$$

◇

Proof Recall the definition of set map (Definition 2.6) and quotient set map (Definition 3.3). δ is a partial function from $\mathit{TypeVar}$. $\delta/\sim$ consists of maplets whose first elements are identical to those of δ. It thus follows that $\delta/\sim$ is also a partial function from $\mathit{TypeVar}$ and $\mathrm{dom}(\delta/\sim) = \mathrm{dom}\, \delta$.

Recall also that δ is a partial function to $\mathbb{P}\, \mathit{Atom}$. Let $\mathit{maplet} : \delta$. It follows that:

$$\begin{array}{l} \mathit{second}(\mathit{maplet}) \in \mathbb{P}\, \mathit{Atom} \\ \Rightarrow \\ (\forall\, x : \mathit{second}(\mathit{maplet}) \bullet x \in \mathit{Atom}) \\ \Rightarrow \\ (\forall\, x : \mathit{second}(\mathit{maplet}) \bullet [x]_\sim \in \mathit{Atom}^\sim) \\ \Rightarrow \\ \{x : \mathit{second}(\mathit{maplet}) \bullet [x]_\sim\} \in \mathbb{P}\, \mathit{Atom}^\sim \end{array} \qquad [3.2]$$

This shows that $\delta/\sim$ is a partial function to $\mathbb{P}\, \mathit{Atom}^\sim$, and completes the proof of the first two consequents of the lemma. To prove the final one we just need

to show:

$$\forall X, Y : \mathrm{dom}(\delta/\sim) \mid X \neq Y \bullet (\delta/\sim)(X) \cap (\delta/\sim)(Y) = \emptyset$$

Suppose for a contradiction the negation of the above. Choose $X, Y : \mathrm{dom}(\delta/\sim)$ such that $X \neq Y$ and $(\delta/\sim)(X) \cap (\delta/\sim)(Y) \neq \emptyset$. Then choose $z' : (\delta/\sim)(X) \cap (\delta/\sim)(X)$. Finally choose $x : \delta(X)$ such that $z' = [x]_\sim$ and $y : \delta(Y)$ such that $z' = [y]_\sim$. It follows that $x \sim y$, contradicting the definition of data relation.

□

We now define a *closed* binding to be one in which every value is a closed value. We will shortly prove that taking the quotient of a binding which is a closed binding gives the expected result i.e. a quotient binding.

Definition 3.23 Closed bindings

Let (Υ, Γ) be a schema and let $\delta : \mathit{SetMap}(\Upsilon)$. Let $\sim : \mathit{DataRel}(\delta)$. Let $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$.

We define $\mathit{Closed}_\sim(\eta)$ as follows:

$$\mathit{Closed}_\sim(\eta) \Leftrightarrow (\forall\, var : \mathrm{dom}\, \eta \bullet \mathit{Closed}_\sim(\eta(var)))$$

where $\mathit{Closed}_\sim(\eta(var))$ was defined in Definition 3.16. ◇

The following theorem tells us that if a value belongs to the semantics of a type with respect to a set map, then the quotient of the value belongs to the semantics of the type with respect to the corresponding quotient set map, provided the value is closed. We will use this theorem to ensure the quotient values in the quotient of a binding are of the correct type.

Theorem 3.24

Let T be a type expression. Let δ be a set map such that:

$$\mathit{Used}(T) \subseteq \mathrm{dom}\, \delta$$

Let $V : [\![T]\!]_\delta$. Let $\sim$ be a data relation. Suppose $\mathit{Closed}_\sim(V)$. Then:

$$V/\sim \in [\![T]\!]_{\delta/\sim}$$

◇

Proof See Theorem A.8. □

Remark 3.25

We need the value to be closed in order to ensure that the quotient of the value satisfies the same multiplicity restrictions. For example, if we use the assignments in Example 3.10 and let:

$$V = \{ \quad \langle \mathit{london}, \mathit{manchester} \rangle, \\ \langle \mathit{manchester}, \mathit{new\ york} \rangle \}$$

Then $V \in [\![\texttt{City set->lone City}]\!]_\delta$ i.e. V represents a partial function from `City` to `City`. However:

$$V/\sim = \{ \quad \langle \{london, manchester\}, \{london, manchester\} \rangle \\ \langle \{london, manchester\}, \{new\ york, philadelphia\} \rangle$$

and $V \notin [\![\texttt{City set->lone City}]\!]_{\delta/\sim}$ i.e. $V/\sim$ does not represent a partial function from `City` to `City`. $\diamond$

We can now prove that the quotient of a binding is a quotient binding, and hence use it in evaluating the semantics of a formula or expression.

Lemma 3.26 The quotient of a binding, which is closed, is a quotient binding.

Let (Υ, Γ) be a schema and let $\delta : \mathit{SetMap}(\Upsilon)$. Let $\sim : \mathit{DataRel}(\delta)$. Let $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$. Suppose $\mathit{Closed}_\sim(\eta)$. Then:

$$\eta/\sim \in \mathit{Binding}^\sim(\Upsilon, \Gamma, \delta/\sim)$$

$\diamond$

Proof Using Definition 3.20 and Definition 2.27 we know: $\mathrm{dom}(\eta/\sim) = \mathrm{dom}\,\eta = \mathrm{dom}\,\Gamma$. Let $var : \mathrm{dom}(\eta/\sim)$. Using Definition 2.27 and Definition 3.23 it follows:

$$\begin{array}{ll} & \eta(var) \in [\![\Gamma(var)]\!]_\delta \wedge \mathit{Closed}_\sim(\eta(var)) \\ \Rightarrow & \\ & \eta(var)/\sim \in [\![\Gamma(var)]\!]_{\delta/\sim} \end{array} \qquad [3.24]$$

Hence by Definition 3.20:

$$\begin{array}{ll} & \forall\, var : \mathrm{dom}(\eta/\sim) \bullet (\eta/\sim)(var) \in [\![\Gamma(var)]\!]_{\delta/\sim} \\ \Rightarrow & \\ & \eta/\sim \in \mathit{Binding}^\sim(\Upsilon, \Gamma, \delta/\sim) \end{array} \qquad [3.4]$$

$\square$

Remark 3.27

Note that in our notation we have overloaded the operation $/\sim$. When applied to values this is the lower adjoint of the Galois connection, but it can be applied to bindings and set maps as well. Thus it is a function between the following:

domain	range
$\mathbb{P}(\mathrm{seq}\, Atom)$	$\mathbb{P}(\mathrm{seq}\, Atom')$
$\mathit{TypeVar} \nrightarrow \mathbb{P}\, Atom$	$\mathit{TypeVar} \nrightarrow \mathbb{P}\, Atom'$
$Var \nrightarrow \mathbb{P}(\mathrm{seq}\, Atom)$	$Var \nrightarrow \mathbb{P}(\mathrm{seq}\, Atom')$

The theory developed in this thesis (e.g. Theorem 3.24) depends on the relationship between these functions, so goes beyond the theory of Galois connections.

◇

3.4 Semantic data independence

Finally we are in a position to give a semantic definition of data independence. To allow us to consider data independence in a single type variable we place a restriction on data relations so that they may equate atoms in the carrier set of the data independent type only.

Definition 3.28 Single type variable data relation

Let $\delta : SetMap$ and $X : \operatorname{dom} \delta$. We define $DataRel_X(\delta)$ as follows:

$$DataRel_X(\delta) = \{\sim : DataRel(\delta) \mid \forall\, x, y : Atom \bullet \\ x \sim y \Rightarrow x \in \delta(X) \vee x = y\}$$

◇

We can now define semantic data independence.

Definition 3.29 Semantic data independence

Let $X : TypeVar$. Let e be an expression. (Respectively let F be a formula). We say e (respectively F) is data independent in X and write $DataInd_X(e)$ (respectively $DataInd_X(F)$) exactly when the following holds:

Given any schema (Υ, Γ) such that $\Gamma \vdash e$ (respectively $\Gamma \vdash F$), given $\delta : SetMap(\Upsilon)$ and $\eta : Binding(\Upsilon, \Gamma, \delta)$, and given $\sim : DataRel_X(\delta)$ such that $Closed_\sim(\eta)$:

$$[\![e]\!]^\eta_\delta = ([\![e]\!]^{\eta/\sim}_{\delta/\sim}).\sim$$

(respectively

$$[\![F]\!]^\eta_\delta \Leftrightarrow [\![F]\!]^{\eta/\sim}_{\delta/\sim}$$

)

◇

Thus an expression is semantically data independent if evaluating it with respect to a closed binding results in the same value as evaluating it with respect to a corresponding quotient binding and converting the resultant quotient value back in to a value.

Example 3.30

We will now give an example of a formula which satisfies the definition of semantic data independence and one which does not.

Let

$$\Upsilon = \{\texttt{City}\}$$
$$\Gamma = \{\texttt{usa} \mapsto \texttt{set City}\}$$

Let F be the following formula:

```
usa = City
```

Then $\Gamma \vdash F$.

We reuse the assignments of Example 3.10 and in addition let:

$$\eta = \{\texttt{usa} \mapsto \{\langle new\ york\rangle, \langle philadelphia\rangle\}\}$$

so that η is a binding for F i.e. $\eta \in Binding(\Upsilon, \Gamma, \delta)$. We can see that η is closed because:

$$(\{\langle new\ york\rangle, \langle philadelphia\rangle\}/\sim).\sim$$
$$=$$
$$\{\langle\{new\ york, philadelphia\}\rangle\}\}.\sim$$
$$=$$
$$\{\langle new\ york\rangle, \langle philadelphia\rangle\}$$

In Chapter 4 we prove that F is a data-independent formula. Since η is closed, we can apply the definition of semantic data independence and expect $[\![F]\!]^{\eta}_{\delta}$ to hold if and only if $[\![F]\!]^{\eta/\sim}_{\delta/\sim}$ holds. This is indeed the case:

$$[\![F]\!]^{\eta}_{\delta}$$
$$\Leftrightarrow$$
$$\{\langle new\ york\rangle, \langle philadelphia\rangle\} =$$
$$\quad \{\langle london\rangle, \langle manchester\rangle, \langle new\ york\rangle, \langle philadelphia\rangle\}$$
$$\Leftrightarrow$$
$$false$$

$$[\![F]\!]^{\eta/\sim}_{\delta/\sim}$$
$$\Leftrightarrow$$
$$\{\langle\{new\ york, philadelphia\}\rangle\} =$$
$$\quad \{\langle\{london, manchester\}\rangle, \langle\{new\ york, philadelphia\}\rangle\}$$
$$\Leftrightarrow$$
$$false$$

Let G be the following formula:

```
some c:City|c = usa
```

It follows:

$$
\begin{array}{l}
[\![G]\!]^{\eta}_{\delta} \\
\Leftrightarrow \\
\exists\, x : \{london, manchester, new\ york, philadelphia\} \bullet \\
\quad \{\langle x \rangle\} = \{\langle new\ york \rangle, \langle philadelphia \rangle\} \\
\Leftrightarrow \\
false
\end{array}
$$

but:

$$
\begin{array}{l}
[\![G]\!]^{\eta/\sim}_{\delta/\sim} \\
\Leftrightarrow \\
\exists\, x : \{\{london, manchester\}, \{new\ york, philadelphia\}\} \bullet \\
\quad \{\langle x \rangle\} = \{\langle \{new\ york, philadelphia\} \rangle\} \\
\Leftrightarrow true
\end{array}
$$

so it is clear G is not data independent. $\diamond$

Summary

In this chapter we introduced an equivalence relation on the uninterpreted set of atoms which is used as the basis for the language semantics. We used this to define a projection between values (sets of sequences of atoms) and quotient values, and showed that this gives rise to a Galois insertion.

We lifted this projection to bindings and called the image of a binding under this projection, the quotient of a binding. We showed how the quotient of a binding could be considered a binding in its own right (for a correct choice of equivalence relation) and therefore used to evaluate the truth of a formula.

We then gave a semantic definition of data independence for a formula, based on whether a quotient binding can be used instead of a binding to evaluate the truth of a formula.

Chapter 5 will concern itself with how to choose an equivalence relation so that the quotient bindings lie within a small scope. Before that the next chapter gives a means to calculate if a formula satisfies the semantic definition.

Chapter 4

Syntactic data independence

The semantic definition of data independence given in the previous chapter is of theoretical importance. It allows one to determine the meaning of a formula or expression for a particular binding using a quotient binding instead. However, we have no automatic test of whether a formula satisfies the semantic definition and therefore it is of limited practical use.

In this chapter we give a syntactic definition of data independence. Testing whether a formula satisfies this definition is just a matter of inspecting which language constructors it uses. A formula which satisfies the syntactic definition is guaranteed to satisfy the semantic one. The converse does not hold, but many formulas satisfy the syntactic definition thus in practice it is a useful definition.

4.1 Data-independent constructors

In this section we introduce the concept of *data-independent constructors*. We prove all the constructors in the language which are data independent to be so. (Those which are not are dealt with in the next section). Then we define syntactically-data-independent formulas as ones which use only the data-independent constructors. We show that formulas, which satisfy the syntactic definition, also satisfy the semantic definition, but not the converse.

Definition 4.1 Data-independent constructors

A constructor of the language is data independent in a type variable X if it always yields an expression/formula which is semantically data independent in X when its operands (if any) are semantically data independent in X. If a constructor is data independent in X for any $X : \mathit{TypeVar}$ we speak of the constructor being data independent.

For example, if $\otimes : \mathit{Expr} \times \mathit{Expr} \to \mathit{Expr}$ is an binary constructor, then $\otimes$

is data independent in X exactly when:

$$\forall\, e, f : Expr \mid DataInd_X(e) \wedge DataInd_X(f) \bullet DataInd_X(e \otimes f)$$

$\diamond$

For each data-independent language constructor, we now give a theorem showing it is data independent. We first prove a subsidiary lemma.

Lemma 4.2
Let $\sim : EqRel$. Let $xs : \mathrm{seq}\, Atom$. Let $W' : \mathrm{seq}\, Atom^{\sim}$. Then:

$$xs/\!\sim \in W' \Leftrightarrow xs \in W'.\sim$$

$\diamond$

Proof

$xs/\!\sim \in W'$
$\Rightarrow$ [3.9]
$(xs/\!\sim).\sim \subseteq W'.\sim$
$\Rightarrow$ [3.11]
$xs \in (xs/\!\sim).\sim \subseteq W'.\sim$
$\Rightarrow$
$xs \in W'.\sim$

$xs \in W'.\sim$
$\Rightarrow$ [3.8]
$xs/\!\sim \in (W'.\sim)/\!\sim$
$\Rightarrow$ [3.14]
$xs/\!\sim \in W'$

$\square$

4.1.1 The ˜ and ˆ constructors

Lemma 4.3
Let $xs : \mathrm{seq}\, Atom$. Let $\sim : EqRel$. Then:

$$rev(xs)/\!\sim = rev(xs/\!\sim)$$

$\diamond$

Proof The proof uses induction over the structure of xs.
Case $\langle\rangle$.

$$
\begin{array}{ll}
& rev(\langle\rangle)/\sim \\
= & \\
& \langle\rangle/\sim \\
= & [3.8] \\
& \langle\rangle \\
= & \\
& rev(\langle\rangle) \\
= & [3.8] \\
& rev(\langle\rangle/\sim)
\end{array}
$$

Case $\langle x\rangle \frown xs$.

$$
\begin{array}{ll}
& rev(\langle x\rangle \frown xs)/\sim \\
= & \\
& (rev(xs) \frown \langle x\rangle)/\sim \\
= & [3.8] \\
& (rev(xs)/\sim) \frown (\langle x\rangle/\sim) \\
= & [\text{ind. hyp.}] \\
& rev(xs/\sim) \frown (\langle x\rangle/\sim) \\
= & \\
& rev((\langle x\rangle/\sim) \frown (xs/\sim)) \\
= & [3.8] \\
& rev((\langle x\rangle \frown xs)/\sim)
\end{array}
$$

□

Lemma 4.4
Let $\sim : EqRel$ and $ys' : \operatorname{seq} Atom^{\sim}$. Then:

$$ys'.\sim \neq \emptyset$$

◇

Proof The proof uses induction on the structure of ys'.
Case $\langle\rangle$.

$$
\begin{array}{ll}
& \langle\rangle.\sim \\
= & [3.9] \\
& \{\langle\rangle\} \\
\neq & \\
& \emptyset
\end{array}
$$

Case $\langle y' \rangle \frown ys'$.
Using Definition 3.2 we first note:

$$
\begin{array}{l}
Atom^{\sim} = \{x : Atom \bullet [x]_{\sim}\} \\
\Rightarrow \\
(\forall\, x' : Atom^{\sim} \bullet x' \neq \emptyset)
\end{array}
$$

It then follows:

$$
\begin{array}{ll}
(\langle y' \rangle \frown ys').\sim & \\
= & [3.9] \\
\{y : y';\ ys : ys'.\sim \bullet \langle y \rangle \frown ys\} & \\
\neq & \\
\emptyset &
\end{array}
$$

since $y' \neq \emptyset$ and ind. hyp. gives $ys'.\sim \neq \emptyset$. □

Theorem 4.5 The $\tilde{}$ and $\hat{}$ constructors are data independent. ◇

Proof We prove the $\tilde{}$ constructor is data independent. The proof for the $\hat{}$ constructor is the same, but one should use Lemma 4.7 in place of Lemma 4.6.

Let $X : TypeVar$. Let $e : Expr$ and suppose $DataInd_X(e)$. To complete the proof we must show $DataInd_X(\tilde{}\ e)$. Let (Υ, Γ) be a schema and suppose $\Gamma \vdash \tilde{}\ e$. Let $\delta : SetMap(\Upsilon)$. Let $\eta : Binding(\Upsilon, \Gamma, \delta)$. Let $\sim : DataRel_X(\delta)$. Suppose $Closed_{\sim}(\eta)$.

$$
\begin{array}{ll}
\Gamma \vdash \tilde{}\ e & \\
\Rightarrow & [2.31] \\
\Gamma \vdash e & \\
\Rightarrow & [3.29] \\
[\![e]\!]^{\eta}_{\delta} = ([\![e]\!]^{\eta/\sim}_{\delta/\sim}).\sim & \\
\Rightarrow & [4.6] \\
[\![\tilde{}\ e]\!]^{\eta}_{\delta} = ([\![\tilde{}\ e]\!]^{\eta/\sim}_{\delta/\sim}).\sim &
\end{array}
$$

Hence $DataInd_X(\tilde{}\ e)$. □

Lemma 4.6
Let $X : TypeVar$. Let $e : Expr$. Let (Υ, Γ) be a schema. Suppose $\Gamma \vdash \tilde{}\ e$. Let $\delta : SetMap(\Upsilon)$. Let $\eta : Binding(\Upsilon, \Gamma, \delta)$. Let $\sim : DataRel_X(\delta)$. Suppose:

$$[\![e]\!]^{\eta}_{\delta} = ([\![e]\!]^{\eta/\sim}_{\delta/\sim}).\sim$$

Then:

$$[\![\tilde{}\ e]\!]^{\eta}_{\delta} = ([\![\tilde{}\ e]\!]^{\eta/\sim}_{\delta/\sim}).\sim$$

$\diamond$

Proof Let $xs : \mathrm{seq}\, Atom$.

$$\begin{array}{ll}
& xs = rev(rev(xs)) \in [\![\tilde{}\ e]\!]^{\eta}_{\delta} \\
\Leftrightarrow & [2.33] \\
& rev(xs) \in [\![e]\!]^{\eta}_{\delta} \\
\Leftrightarrow & [3.8,\ 3.14] \\
& rev(xs)/\sim \in [\![e]\!]^{\eta}_{\delta}/\sim = (([\![e]\!]^{\eta/\sim}_{\delta/\sim}).\sim)/\sim = [\![e]\!]^{\eta/\sim}_{\delta/\sim} \\
\Leftrightarrow & [4.3] \\
& rev(xs/\sim) \in [\![e]\!]^{\eta/\sim}_{\delta/\sim} \\
\Leftrightarrow & [2.33] \\
& xs/\sim \in [\![\tilde{}\ e]\!]^{\eta/\sim}_{\delta/\sim} \\
\Leftrightarrow & [4.2] \\
& xs \in ([\![\tilde{}\ e]\!]^{\eta/\sim}_{\delta/\sim}).\sim
\end{array}$$

$\square$

Lemma 4.7

Let $X : \mathit{TypeVar}$. Let $e : \mathit{Expr}$. Let (Υ, Γ) be a schema. Suppose $\Gamma \vdash \hat{}\ e$. Let $\delta : \mathit{SetMap}(\Upsilon)$. Let $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$. Let $\sim : \mathit{DataRel}_X(\delta)$. Suppose:

$$[\![e]\!]^{\eta}_{\delta} = ([\![e]\!]^{\eta/\sim}_{\delta/\sim}).\sim$$

Then:

$$[\![\hat{}\ e]\!]^{\eta}_{\delta} = ([\![\hat{}\ e]\!]^{\eta/\sim}_{\delta/\sim}).\sim$$

$\diamond$

Proof We show $[\![\hat{}\ e]\!]^{\eta}_{\delta} = ([\![\hat{}\ e]\!]^{\eta/\sim}_{\delta/\sim}).\sim$ by showing inclusion in both directions. Considering Definition 2.33 it is clear both sides of the above equality contain lists of length 2 only (if any). Let $x, z : \mathit{Atom}$.

Suppose $\langle x, z\rangle \in [\![\hat{}\ e]\!]^{\eta}_{\delta}$. Choose $ys : \mathrm{seq}\, Atom$ such that:

$$\begin{array}{l}
x = head(ys) \\
z = last(ys) \\
\forall\, i : 1 \mathinner{..} \#(ys) - 1 \bullet \langle ys(i), ys(i+1)\rangle \in [\![e]\!]^{\eta}_{\delta}
\end{array}$$

Let $ys' = ys/\sim$. Using Lemma 4.2 and $[\![e]\!]^\eta_\delta = ([\![e]\!]^{\eta/\sim}_{\delta/\sim}).\sim$ it follows:

$$\forall\, i : 1\,..\,\#(ys') - 1 \bullet \langle ys'(i), ys'(i+1)\rangle \in [\![e]\!]^{\eta/\sim}_{\delta/\sim}$$

Thus:

$$\begin{array}{ll} & \langle x, z\rangle/\sim = \langle head(ys'), last(ys')\rangle \in [\![\hat{}\ e]\!]^{\eta/\sim}_{\delta/\sim} \\ & \Rightarrow \qquad\qquad [4.2] \\ & \langle x, z\rangle \in ([\![\hat{}\ e]\!]^{\eta/\sim}_{\delta/\sim}).\sim \end{array}$$

This proves the forwards inclusion.

For the reverse inclusion, now suppose $\langle x, z\rangle \in ([\![\hat{}\ e]\!]^{\eta/\sim}_{\delta/\sim}).\sim$. By Lemma 4.2:

$$\langle x, z\rangle/\sim = \langle [x]_\sim, [z]_\sim\rangle \in [\![\hat{}\ e]\!]^{\eta/\sim}_{\delta/\sim}$$

Choose $ys' : \operatorname{seq} Atom^\sim$ such that:

$$\begin{array}{l} [x]_\sim = head(ys') \\ [z]_\sim = last(ys') \\ \forall\, i : 1\,..\,\#(ys) - 1 \bullet \langle ys'(i), ys'(i+1)\rangle \in [\![e]\!]^{\eta/\sim}_{\delta/\sim} \end{array}$$

Then choose $ys : ys'.\sim$, which is non-empty by Lemma 4.4. Then $\forall\, i : 1..\#(ys) \bullet ys(i) \in ys'(i)$. So:

$$\langle x, head(ys)\rangle \in \langle [x]_\sim, head(ys')\rangle.\sim \subseteq [\![e]\!]^{\eta/\sim}_{\delta/\sim}.\sim = [\![e]\!]^\eta_\delta$$

Similarly:

$$\begin{array}{l} \forall\, i : 1\,..\,\#(ys) - 1 \bullet \\ \qquad \langle ys(i), ys(i+1)\rangle \in \langle ys'(i), ys'(i+1)\rangle.\sim \subseteq [\![e]\!]^{\eta/\sim}_{\delta/\sim}.\sim = [\![e]\!]^\eta_\delta \end{array}$$

Hence:

$$\langle x, z\rangle = \langle head(ys), last(ys)\rangle \in [\![\hat{}\ e]\!]^\eta_\delta$$

This completes the reverse inclusion. □

4.1.2 The +, -, &, ., and -> constructors

Theorem 4.8 The following constructors are data independent: `+`, `-`, `&`, `.`, `->`. ◇

Proof We prove the case `-`. The other cases are very similar. Lemma 4.10 should be used in place of Lemma 4.9 for the last two cases.

Let $X : \mathit{TypeVar}$. Let $e, f : \mathit{Expr}$ and suppose $\mathit{DataInd}_X(e) \wedge \mathit{DataInd}_X(f)$. To complete the proof we must show $\mathit{DataInd}_X(e \texttt{ - } f)$. Let (Υ, Γ) be a schema and suppose $\Gamma \vdash e \texttt{ - } f$. Let $\delta : \mathit{SetMap}(\Upsilon)$. Let $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$. Let $\sim : \mathit{DataRel}_X(\delta)$. Suppose $\mathit{Closed}_\sim(\eta)$.

$$
\begin{array}{ll}
& \Gamma \vdash e \texttt{ - } f \\
\Rightarrow & [2.31] \\
& \Gamma \vdash e \wedge \Gamma \vdash f \\
\Rightarrow & [3.29] \\
& [\![e]\!]^\eta_\delta = ([\![e]\!]^{\eta/\sim}_{\delta/\sim}).\sim \wedge [\![f]\!]^\eta_\delta = ([\![f]\!]^{\eta/\sim}_{\delta/\sim}).\sim \\
\Rightarrow & [4.9] \\
& [\![e \texttt{ - } f]\!]^\eta_\delta = ([\![e \texttt{ - } f]\!]^{\eta/\sim}_{\delta/\sim}).\sim
\end{array}
$$

Hence $\mathit{DataInd}_X(e \texttt{ - } f)$. □

Lemma 4.9

Let $\otimes$ stand for one of the following constructors: `+`, `-`, `&`. Let $X : \mathit{TypeVar}$. Let $e, f : \mathit{Expr}$. Let (Υ, Γ) be a schema and suppose $\Gamma \vdash e \otimes f$. Let $\delta : \mathit{SetMap}(\Upsilon)$. Let $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$. Let $\sim : \mathit{DataRel}_X(\delta)$. Suppose $\mathit{Closed}_\sim(\eta)$. Suppose:

$$[\![e]\!]^\eta_\delta = ([\![e]\!]^{\eta/\sim}_{\delta/\sim}).\sim \wedge [\![f]\!]^\eta_\delta = ([\![f]\!]^{\eta/\sim}_{\delta/\sim}).\sim$$

Then:

$$[\![e \otimes f]\!]^\eta_\delta = ([\![e \otimes f]\!]^{\eta/\sim}_{\delta/\sim}).\sim$$

◇

Proof We will prove the case when $\otimes$ stands for `-` only, as the other cases follow similarly.

Let $xs : \mathrm{seq}\, \mathit{Atom}$.

$$
\begin{array}{ll}
& xs \in [\![e \texttt{ - } f]\!]^\eta_\delta \\
\Leftrightarrow & [2.33] \\
& xs \in [\![e]\!]^\eta_\delta \wedge \neg\, xs \in [\![f]\!]^\eta_\delta \\
\Leftrightarrow & \\
& xs \in ([\![e]\!]^{\eta/\sim}_{\delta/\sim}).\sim \wedge \neg\, xs \in ([\![f]\!]^{\eta/\sim}_{\delta/\sim}).\sim \\
\Leftrightarrow & [4.2] \\
& xs/\sim \in [\![e]\!]^{\eta/\sim}_{\delta/\sim} \wedge \neg\, xs/\sim \in [\![f]\!]^{\eta/\sim}_{\delta/\sim} \\
\Leftrightarrow & [2.33] \\
& xs/\sim \in [\![e \texttt{ - } f]\!]^{\eta/\sim}_{\delta/\sim}
\end{array}
$$

$\Leftrightarrow$ [4.2]

$xs \in ([\![e \ \texttt{-} \ f]\!]^{\eta/\sim}_{\delta/\sim}).\sim$

□

Lemma 4.10

Let $X : \mathit{TypeVar}$. Let $e, f : \mathit{Expr}$. Let $\otimes$ stand for: `.` or `->`. Let (Υ, Γ) be a schema, suppose that $\Gamma \vdash e \ . \ f$. Let $\delta : \mathit{SetMap}(\Upsilon)$. Let $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$. Let $\sim \mathit{DataRel}_X(\delta)$. Suppose $\mathit{Closed}_\sim(\eta)$. Suppose:

$$[\![e]\!]^\eta_\delta = ([\![e]\!]^{\eta/\sim}_{\delta/\sim}).\sim \wedge [\![f]\!]^\eta_\delta = ([\![f]\!]^{\eta/\sim}_{\delta/\sim}).\sim$$

Then:

$$[\![e \otimes f]\!]^\eta_\delta = ([\![e \otimes f]\!]^{\eta/\sim}_{\delta/\sim}).\sim$$

◇

Proof We will prove the case when $\otimes$ stands for `.` only, as the other case is very similar. We show:

$$[\![e \ . \ f]\!]^\eta_\delta = ([\![e \ . \ f]\!]^{\eta/\sim}_{\delta/\sim}).\sim$$

by showing inclusion in both directions.

Suppose $zs : [\![e \ . \ f]\!]^\eta_\delta$. Using Definition 2.33, choose $xs : [\![e]\!]^\eta_\delta$ and $ys : [\![f]\!]^\eta_\delta$ such that:

$$\mathit{last}(xs) = \mathit{head}(ys) \wedge zs = \mathit{front}(xs) \frown \mathit{tail}(ys)$$

Put $xs' = xs/\sim$, $ys' = ys/\sim$, $zs' = zs/\sim$. Then:

$$\mathit{last}(xs') = [\mathit{last}(xs)]_\sim = [\mathit{head}(ys)]_\sim = \mathit{head}(ys')$$
$$zs' = \mathit{front}(xs') \frown \mathit{tail}(ys')$$

and:

$xs \in ([\![e]\!]^{\eta/\sim}_{\delta/\sim}).\sim$

$\Rightarrow$ [4.2]

$xs' \in [\![e]\!]^{\eta/\sim}_{\delta/\sim}$

Similarly:

$$ys' \in [\![f]\!]^{\eta/\sim}_{\delta/\sim}$$

Hence:

$$zs' \in [\![e \ . \ f]\!]^{\eta/\sim}_{\delta/\sim}$$

$$
\begin{array}{l}
\Rightarrow \\
zs/\sim \in [\![e \ . \ f]\!]^{\eta/\sim}_{\delta/\sim} \\
\Rightarrow \\
zs \in [\![e \ . \ f]\!]^{\eta/\sim}_{\delta/\sim}.\sim
\end{array}
\qquad [4.2]
$$

This completes the proof of the forwards inclusion.

Now let $zs : ([\![e \ . \ f]\!]^{\eta/\sim}_{\delta/\sim}).\sim$. It follows:

$$zs/\sim \in [\![e \ . \ f]\!]^{\eta/\sim}_{\delta/\sim}$$

Choose $xs' : [\![e]\!]^{\eta/\sim}_{\delta/\sim}$ and $ys' : [\![f]\!]^{\eta/\sim}_{\delta/\sim}$ such that:

$$
\begin{array}{l}
zs/\sim = \mathit{front}(xs') \frown \mathit{tail}(ys') \\
\mathit{last}(xs') = \mathit{head}(ys')
\end{array}
$$

Let $w : \mathit{last}(xs')$. Define $xs, ys : \mathrm{seq}\, Atom$ as follows:

$$
\begin{array}{l}
\mathrm{dom}\, xs = \mathrm{dom}\, xs' \\
xs(i) = \textbf{if } (i < \#(xs')) \textbf{ then } zs(i) \textbf{ else } w \\
\mathrm{dom}\, ys = \mathrm{dom}\, ys' \\
ys(i) = \textbf{if } (i > 1) \textbf{ then } zs(\#(xs') + i - 3) \textbf{ else } w
\end{array}
$$

It follows:

$$
\begin{array}{l}
zs = \mathit{front}(xs) \frown \mathit{tail}(ys) \\
\mathit{last}(xs) = \mathit{head}(ys) \\
xs/\sim = xs' \\
ys/\sim = ys'
\end{array}
$$

Hence:

$$
\begin{array}{l}
xs \in [\![e]\!]^{\eta/\sim}_{\delta/\sim}.\sim = [\![e]\!]^{\eta}_{\delta} \\
ys \in [\![f]\!]^{\eta/\sim}_{\delta/\sim}.\sim = [\![f]\!]^{\eta}_{\delta}
\end{array}
$$

Thus:

$$zs \in [\![e \ . \ f]\!]^{\eta}_{\delta}$$

□

4.1.3 The `in` and `=` constructors

Theorem 4.11 The following constructors are data independent: `in`, `=`. ◇

Proof We prove the case `in` only, as the case `=` is very similar. Let X : *TypeVar*. Let e, f : *Expr* and suppose $\mathit{DataInd}_X(e) \wedge \mathit{DataInd}_X(f)$. We want to show $\mathit{DataInd}_X(e \texttt{ in } f)$. Let (Υ, Γ) be a schema, suppose that $\Gamma \vdash e \texttt{ in } f$. Let δ : $\mathit{SetMap}(\Upsilon)$. Let η : $\mathit{Binding}(\Upsilon, \Gamma, \delta)$. Let $\sim$: $\mathit{DataRel}_X(\delta)$. Suppose $\mathit{Closed}_\sim(\eta)$.

$$\begin{array}{ll} & \Gamma \vdash e \texttt{ in } f \\ \Rightarrow & \\ & \Gamma \vdash e \wedge \Gamma \vdash f \\ \Rightarrow & [3.29] \\ & [\![e]\!]^\eta_\delta = ([\![e]\!]^{\eta/\sim}_{\delta/\sim}).\sim \wedge [\![f]\!]^\eta_\delta = ([\![f]\!]^{\eta/\sim}_{\delta/\sim}).\sim \\ \Rightarrow & [4.12] \\ & [\![e \texttt{ in } f]\!]^\eta_\delta \Leftrightarrow [\![e \texttt{ in } f]\!]^{\eta/\sim}_{\delta/\sim} \end{array}$$

Hence $\mathit{DataInd}_X(e \texttt{ in } f)$. □

Lemma 4.12

Let $\otimes$ stand for `in` or `=`. Let X : *TypeVar*. Let e, f : *Expr*. Let (Υ, Γ) be a schema, suppose that $\Gamma \vdash e \otimes f$. Let δ : $\mathit{SetMap}(\Upsilon)$. Let η : $\mathit{Binding}(\Upsilon, \Gamma, \delta)$. Let $\sim$: $\mathit{DataRel}_X(\delta)$. Suppose $\mathit{Closed}_\sim(\eta)$. Suppose:

$$[\![e]\!]^\eta_\delta = ([\![e]\!]^{\eta/\sim}_{\delta/\sim}).\sim \wedge [\![f]\!]^\eta_\delta = ([\![f]\!]^{\eta/\sim}_{\delta/\sim}).\sim$$

Then:

$$[\![e \otimes f]\!]^\eta_\delta \Leftrightarrow [\![e \otimes f]\!]^{\eta/\sim}_{\delta/\sim}$$

◇

Proof We prove the case when $\otimes$ stands for `in` only, as the other case is very similar.

We need to show:

$$[\![e \texttt{ in } f]\!]^\eta_\delta \Leftrightarrow [\![e \texttt{ in } f]\!]^{\eta/\sim}_{\delta/\sim}$$

So it is sufficient to show:

$$[\![e]\!]^\eta_\delta \subseteq [\![f]\!]^\eta_\delta \Leftrightarrow [\![e]\!]^{\eta/\sim}_{\delta/\sim} \subseteq [\![f]\!]^{\eta/\sim}_{\delta/\sim}$$

We show implication in both directions:

$$\begin{array}{ll} & [\![e]\!]^{\eta/\sim}_{\delta/\sim} \subseteq [\![f]\!]^{\eta/\sim}_{\delta/\sim} \\ \Rightarrow & [3.9] \\ & ([\![e]\!]^{\eta/\sim}_{\delta/\sim}).\sim \subseteq ([\![f]\!]^{\eta/\sim}_{\delta/\sim}).\sim \end{array}$$

$\Rightarrow$
$[\![e]\!]^{\eta}_{\delta} \subseteq [\![f]\!]^{\eta}_{\delta}$

$[\![e]\!]^{\eta}_{\delta} \subseteq [\![f]\!]^{\eta}_{\delta}$
$\Rightarrow$ [3.9]
$([\![e]\!]^{\eta/\sim}_{\delta/\sim}).\sim \subseteq ([\![f]\!]^{\eta/\sim}_{\delta/\sim}).\sim$
$\Rightarrow$ [3.8]
$(([\![e]\!]^{\eta/\sim}_{\delta/\sim}).\sim)/\sim \subseteq (([\![f]\!]^{\eta/\sim}_{\delta/\sim}).\sim)/\sim$
$\Rightarrow$
$[\![e]\!]^{\eta/\sim}_{\delta/\sim} \subseteq [\![f]\!]^{\eta/\sim}_{\delta/\sim}$

□

4.1.4 The `=>`, `or`, and `and` constructors

Theorem 4.13 The logic operation constructors (`=>`, `or`, `and`) are data independent. ◇

Proof We show the `=>` constructor is data independent; the proof for the others is very similar.

Let X : *TypeVar*. Let F, G : *Formula* and suppose $DataInd_X(F)$ and $DataInd_X(G)$. We want to show $DataInd_X(F \texttt{ => } G)$. Let (Υ, Γ) be a schema, suppose that $\Gamma \vdash F \texttt{ => } G$. Let $\delta : SetMap(\Upsilon)$. Let $\eta : Binding(\Upsilon, \Gamma, \delta)$. Let $\sim : DataRel_X(\delta)$. Suppose $Closed_{\sim}(\eta)$.

$\Gamma \vdash F \texttt{ => } G$
$\Rightarrow$ [2.31]
$\Gamma \vdash F \wedge \Gamma \vdash G$
$\Rightarrow$ [3.29]
$([\![F]\!]^{\eta}_{\delta} \Leftrightarrow [\![F]\!]^{\eta/\sim}_{\delta/\sim} \wedge [\![G]\!]^{\eta}_{\delta} \Leftrightarrow [\![G]\!]^{\eta/\sim}_{\delta/\sim})$
$\Rightarrow$ [4.14]
$[\![F \texttt{ => } G]\!]^{\eta}_{\delta} \Leftrightarrow [\![F \texttt{ => } G]\!]^{\eta/\sim}_{\delta/\sim}$

□

Lemma 4.14

Let $\otimes$ stand for one of the following constructors: `=>`, `or`, `and`.

Let X : *TypeVar*. Let F, G : *Formula* Let (Υ, Γ) be a schema, suppose that $\Gamma \vdash F \otimes G$. Let $\delta : SetMap(\Upsilon)$. Let $\eta : Binding(\Upsilon, \Gamma, \delta)$. Let $\sim DataRel_X(\delta)$.

Suppose $\mathit{Closed}_{\sim}(\eta)$. Suppose:

$$[\![F]\!]^{\eta}_{\delta} \Leftrightarrow [\![F]\!]^{\eta/\sim}_{\delta/\sim} \wedge [\![G]\!]^{\eta}_{\delta} \Leftrightarrow [\![G]\!]^{\eta/\sim}_{\delta/\sim}$$

Then:

$$[\![F \otimes G]\!]^{\eta}_{\delta} \Leftrightarrow [\![F \otimes G]\!]^{\eta/\sim}_{\delta/\sim}$$

◇

Proof We will prove the case when $\otimes$ stands for `=>` only, as the other cases are very similar.

Suppose:

$$[\![F]\!]^{\eta}_{\delta} \Leftrightarrow [\![F]\!]^{\eta/\sim}_{\delta/\sim} \wedge [\![G]\!]^{\eta}_{\delta} \Leftrightarrow [\![G]\!]^{\eta/\sim}_{\delta/\sim}$$

It follows:

$$\begin{array}{ll}
& [\![F \texttt{ => } G]\!]^{\eta}_{\delta} \\
\Leftrightarrow & [2.33] \\
& ([\![F]\!]^{\eta}_{\delta} \Rightarrow [\![G]\!]^{\eta}_{\delta}) \\
\Leftrightarrow & \\
& ([\![F]\!]^{\eta/\sim}_{\delta/\sim} \Rightarrow [\![G]\!]^{\eta/\sim}_{\delta/\sim}) \\
\Leftrightarrow & [2.33] \\
& [\![F \texttt{ => } G]\!]^{\eta/\sim}_{\delta/\sim}
\end{array}$$

□

4.1.5 The quantification and set formation constructs

Theorem 4.15 Quantification and set formation over a type variable is data independent in every other type variable.

Let $X : \mathit{TypeVar}$. Let $Y : \mathit{TypeVar} \setminus \{X\}$. Let $\mathit{var} : \mathit{Var}$. The following constructors are data independent in X.

`all` var`:` Y `|` ...
`some` var`:` Y `|` ...
$\{\mathit{var}$`:` Y `|` ...$\}$

◇

Proof

Let $F : \mathit{Formula}$ and suppose $\mathit{DataInd}_X(F)$. We need to show:

$\mathit{DataInd}_X(\texttt{all}\ \mathit{var}\texttt{:}\ Y\ \texttt{|}\ F)$
$\mathit{DataInd}_X(\texttt{some}\ \mathit{var}\texttt{:}\ Y\ \texttt{|}\ F)$
$\mathit{DataInd}_X(\{\mathit{var}\texttt{:}\ Y\ \texttt{|}\ F\})$

So let (Υ, Γ) be a schema such that $\Gamma \vdash \texttt{all}\ \mathit{var}\texttt{:}\ Y\ \texttt{|}\ F$. Let $\delta : \mathit{SetMap}(\Upsilon)$. Let $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$. Let $\sim : \mathit{DataRel}_X(\delta)$ and suppose $\mathit{Closed}_\sim(\eta)$.

It follows $\Gamma \oplus \{\mathit{var} \mapsto (\texttt{set}\ Y)\} \vdash F$. Let $y : \delta(Y)$ and let:

$$\eta_y = \eta \oplus \{\mathit{var} \mapsto \{\langle y \rangle\}\}$$

Because $\sim : \mathit{DataRel}_X(\delta)$ and $Y \neq X$:

$[y]_\sim = \{y\}$
$\Rightarrow$
$\mathit{Closed}_\sim(\{\langle y \rangle\})$
$\Rightarrow$
$\mathit{Closed}_\sim(\eta_y)$

Therefore:

$$[\![F]\!]^{\eta_y}_\delta \Leftrightarrow [\![F]\!]^{\eta_y/\sim}_{\delta/\sim}$$

It follows:

$[\![\ \texttt{all}\ \mathit{var}\texttt{:}\ Y\ \texttt{|}\ F]\!]^{\eta/\sim}_{\delta/\sim}$
$\Leftrightarrow$ [2.33]
$(\forall\, y' : (\delta/\sim)(Y) \bullet [\![F]\!]^{\eta/\sim \oplus \{\mathit{var} \mapsto \{\langle y' \rangle\}\}}_{\delta/\sim})$
$\Leftrightarrow$ [put $\{y\} = y'$]
$(\forall\, y : \delta(Y) \bullet [\![F]\!]^{\eta/\sim \oplus \{\mathit{var} \mapsto \{\langle \{y\} \rangle\}\}}_{\delta/\sim})$
$\Leftrightarrow$
$(\forall\, y : \delta(Y) \bullet [\![F]\!]^{\eta_y/\sim}_{\delta/\sim})$
$\Leftrightarrow$
$(\forall\, y : \delta(Y) \bullet [\![F]\!]^{\eta_y}_{\delta})$
$\Leftrightarrow$ [2.33]
$[\![\ \texttt{all}\ \mathit{var}\texttt{:}\ Y\ \texttt{|}\ F]\!]^{\eta}_{\delta}$

Similarly:

$[\![\ \texttt{some}\ \mathit{var}\texttt{:}\ Y\ \texttt{|}\ F]\!]^{\eta/\sim}_{\delta/\sim}$

$\Leftrightarrow$

$[\![\texttt{some}\ \ var\texttt{:}\ Y \mid F]\!]^{\eta}_{\delta}$

Thus $DataInd_X(\texttt{all}\ var\texttt{:}\ Y \mid F)$ and $DataInd_X(\texttt{some}\ var\texttt{:}\ Y \mid F)$.

We complete the proof by showing $DataInd_X(\{var\texttt{:}\ Y \mid F\})$. Let $y : \delta(Y)$.

$\langle y \rangle \in [\![\{var\texttt{:}\ Y \mid F\}]\!]^{\eta}_{\delta}$

$\Leftrightarrow$ [2.33]

$[\![F]\!]^{\eta_y}_{\delta}$

$\Leftrightarrow$

$[\![F]\!]^{\eta_y/\sim}_{\delta/\sim}$

$\Leftrightarrow$

$[\![F]\!]^{\eta/\sim \oplus \{var \mapsto \{\langle \{y\} \rangle\}\}}_{\delta/\sim}$

$\Leftrightarrow$

$\langle \{y\} \rangle = \langle y \rangle / \sim \in [\![\{var\texttt{:}\ Y \mid F\}]\!]^{\eta/\sim}_{\delta/\sim}$

$\Leftrightarrow$

$\langle y \rangle \in ([\![\{var\texttt{:}\ Y \mid F\}]\!]^{\eta/\sim}_{\delta/\sim}).\sim$

□

We have now proved that all the constructors of the language, except quantification and set formation over the data-independent type variable, are data-independent constructors. In the following section we shall give counterexamples to show that these exceptions are not data-independent constructors. Before that, we give a syntactic definition of data independence and show that formulas that satisfy this definition also satisfy the semantic definition of data independence.

4.1.6 Syntactic data independence

Definition 4.16 Syntactic data independence

A formula or expression is syntactically data independent if it is built using the above data-independent constructors. In other words a formula is syntactically data independent in a type variable X so long as it does not use quantification or set formation constructs over X. ◇

Theorem 4.17 Syntactic data independence implies semantic data independence.

Let X : *TypeVar*. Let F be a formula (respectively let e be an expression) which is syntactically data independent in X. Then F (respectively e) is semantically data independent in X. ◇

Proof The proof uses induction over the structure of the formula or expression. The inductive steps are simply applications of the above theorems regarding data independent constructors. This leaves two bases cases: expressions which are just variables and expressions which are just type variables.

Case *var*.

Let (Υ, Γ) be a schema such that $\Gamma \vdash var$. Let $\delta : SetMap(\Upsilon)$ and $\eta : Binding(\Upsilon, \Gamma, \delta)$. Let $\sim : DataRel_X(\delta)$ such that $Closed_\sim(\eta)$.

$$
\begin{array}{ll}
& Closed_\sim(\eta) \\
\Rightarrow & [3.23] \\
& Closed_\sim(\eta(var)) \\
\Rightarrow & [4.18] \\
& [\![var]\!]^{\eta/\sim}_{\delta/\sim}.\sim = [\![var]\!]^{\eta}_{\delta}
\end{array}
$$

Hence *var* is a data independent expression.

Case Y.

Let (Υ, Γ) be a schema such that $\Gamma \vdash Y$. Let $X : \Upsilon$. Let $\delta : SetMap(\Upsilon)$ and $\eta : Binding(\Upsilon, \Gamma, \delta)$. Let $\sim : DataRel_X(\delta)$ such that $Closed_\sim(\eta)$.

It follows directly from Lemma 4.19 that:

$$[\![Y]\!]^{\eta/\sim}_{\delta/\sim}.\sim = [\![Y]\!]^{\eta}_{\delta}$$

Hence Y is a data independent expression. □

Lemma 4.18

Let (Υ, Γ) be a schema such that $\Gamma \vdash var$. Let $X : \Upsilon$. Let $\delta : SetMap(\Upsilon)$ and $\eta : Binding(\Upsilon, \Gamma, \delta)$. Let $\sim : DataRel_X(\delta)$. Suppose $Closed_\sim(\eta(var))$. Then:

$$[\![var]\!]^{\eta/\sim}_{\delta/\sim}.\sim = [\![var]\!]^{\eta}_{\delta}$$

◇

Proof

$$
\begin{array}{ll}
& Closed_\sim(\eta(var)) \\
\Rightarrow & [3.16] \\
& (\eta(var)/\sim).\sim = \eta(var) \\
\Rightarrow & [3.20] \\
& (\eta/\sim)(var).\sim = \eta(var) \\
\Rightarrow & [2.33] \\
& [\![var]\!]^{\eta/\sim}_{\delta/\sim}.\sim = [\![var]\!]^{\eta}_{\delta}
\end{array}
$$

□

Lemma 4.19

Let (Υ, Γ) be a schema. Let X, $Y : \Upsilon$. Suppose $\Gamma \vdash Y$. Let $\delta : \mathit{SetMap}(\Upsilon)$ and $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$. Let $\sim : \mathit{DataRel}_X(\delta)$. Then:

$$[\![Y]\!]^{\eta/\sim}_{\delta/\sim}.\sim = [\![Y]\!]^{\eta}_{\delta}$$

$\diamond$

Proof Note that:

$$\begin{array}{ll} & [\![Y]\!]^{\eta/\sim}_{\delta/\sim}.\sim \\ = & [2.33] \\ & \{y' : (\delta/\sim)(Y) \bullet \langle y' \rangle\}.\sim \\ = & [3.9] \\ & \bigcup\{y' : (\delta/\sim)(Y) \bullet \langle y' \rangle.\sim\} \\ = & [3.9] \\ & \bigcup\{y' : (\delta/\sim)(Y) \bullet \{y : y' \bullet \langle y \rangle\}\} \end{array}$$

contains nothing but singleton lists of atoms, and:

$$\begin{array}{ll} & [\![Y]\!]^{\eta}_{\delta} \\ = & [2.33] \\ & \{y : \delta(Y) \bullet \langle y \rangle\} \end{array}$$

contains nothing but singleton lists of atoms. We will show:

$$[\![Y]\!]^{\eta/\sim}_{\delta/\sim}.\sim = [\![Y]\!]^{\eta}_{\delta}$$

by showing that every singleton list of atoms in the former is in the latter and vice versa. Let $x : \mathit{Atom}$.

Suppose:

$$\langle x \rangle \in [\![Y]\!]^{\eta/\sim}_{\delta/\sim}.\sim = \bigcup\{y' : (\delta/\sim)(Y) \bullet \{y : y' \bullet \langle y \rangle\}\}$$

Choose $y' : (\delta/\sim)(Y)$ such that $x \in y'$. Recall Definition 3.19 i.e.

$$(\delta/\sim)(Y) = \{y : \delta(Y) \bullet [y]_\sim\}$$

Choose $y : \delta(Y)$ such that $y' = [y]_\sim$. Consequently $x \in [y]_\sim$, thus $x \sim y$. Now $\sim \in \mathit{DataRel}_X(\delta) \subseteq \mathit{DataRel}(\delta)$, and hence by Definition 3.18:

$$\begin{array}{l} x \in \delta(Y) \\ \Rightarrow \\ \langle x \rangle \in [\![Y]\!]^{\eta}_{\delta} \end{array}$$

Now suppose instead:

$$\langle x \rangle \in [\![Y]\!]^{\eta}_{\delta}$$

It follows:

$$x \in [x]_{\sim} \in (\delta/\sim)(Y)$$
$$\Rightarrow$$
$$\langle x \rangle \in \bigcup \{ y' : (\delta/\sim)(Y) \bullet \{ y : y' \bullet \langle y \rangle \} \}$$
$$\Rightarrow$$
$$\langle x \rangle \in [\![Y]\!]^{\eta/\sim}_{\delta/\sim}.\sim$$

□

4.2 Data-dependent constructors

In this section we show that the remaining constructors: quantification and set formation over the data independent type are data dependent constructors (i.e. not data independent constructors).

Theorem 4.20 Quantification and set formation over a type variable is not data independent in that type variable.

Let $X : \mathit{TypeVar}$. Let $\mathit{var} : \mathit{Var}$. The following constructors are not data independent in X.

`all` var`:`X`|` ...
`some` var`:`X`|` ...
`{`var`:`X`|` ...`}`

◇

Proof We will give a counterexample for each constructor. First make the following assignments:

$$\begin{aligned} \Upsilon &= \{X\} \\ \Gamma &= \emptyset \\ \delta &= \{X \mapsto \{x_1, x_2\}\} \\ \eta &= \emptyset \\ \sim &= \delta(X) \times \delta(X) \end{aligned}$$

and it follows: (Υ, Γ) is a schema and:

$$\delta \in \mathit{SetMap}(\Upsilon)$$
$$\eta \in \mathit{Binding}(\Upsilon, \Gamma, \delta)$$
$$\sim \in \mathit{DataRel}_X(\delta)$$
$$\mathit{Closed}_{\sim}(\eta)$$

Now consider the formula:

$$\texttt{all}\ var\texttt{:}X\ \texttt{|}\ var\ \texttt{=}\ X$$

which is compatible with the schema (Υ, Γ). This formula is not data independent in X because:

$$[\![\ \texttt{all}\ var\texttt{:}X\ \texttt{|}\ var\ \texttt{=}\ X]\!]^{\eta}_{\delta} \Leftrightarrow \mathit{false}$$
$$[\![\ \texttt{all}\ var\texttt{:}X\ \texttt{|}\ var\ \texttt{=}\ X]\!]^{\eta/\sim}_{\delta/\sim} \Leftrightarrow \mathit{true}$$

However we have already established that the subexpression:

$$var\ \texttt{=}\ X$$

is syntactically and hence semantically data independent in X. This shows that universal quantification over X is a construct which is not data independent in X.

Now consider the formula:

$$\texttt{some}\ var\texttt{:}X\ \texttt{|}\ var\ \texttt{=}\ X$$

which is compatible with the schema (Υ, Γ). This formula is not data independent in X because:

$$[\![\ \texttt{some}\ var\texttt{:}X\ \texttt{|}\ var\ \texttt{=}\ X]\!]^{\eta}_{\delta} \Leftrightarrow \mathit{false}$$
$$[\![\ \texttt{some}\ var\texttt{:}X\ \texttt{|}\ var\ \texttt{=}\ X]\!]^{\eta/\sim}_{\delta/\sim} \Leftrightarrow \mathit{true}$$

This shows that existential quantification over X is a construct which is not data independent in X, for similar reasons as above.

Now consider the expression:

$$\{var\texttt{:}X\ \texttt{|}\ var\ \texttt{=}\ X\}$$

which is compatible with the schema (Υ, Γ). This formula is not data independent in X because:

$$[\![\{var\texttt{:}X\ \texttt{|}\ var\ \texttt{=}\ X\}]\!]^{\eta}_{\delta} = \{\}$$
$$[\![\{var\texttt{:}X\ \texttt{|}\ var\ \texttt{=}\ X\}]\!]^{\eta/\sim}_{\delta/\sim}.\sim = \{\langle x_1\rangle, \langle x_2\rangle\}$$

This shows that set formation with respect to X is a construct which is not data independent in X, for similar reasons as above. □

Theorem 4.21 Semantic data independence does not in general imply syntactic data independence.

Define the formula F by:

$$F = \texttt{all x:X|x = x}$$

in the context of schema $(\{\mathtt{X}\}, \emptyset)$ i.e. the formula has no free variables and relies on one type variable: $\mathtt{X}$.

Evaluating F under any set map and binding clearly gives the value *true* and as such is it easy to see that F is semantically data independent with respect to $\mathtt{X}$ (Definition 3.29). However F use quantification over the data-independent type, a data-dependent construct, and so F is not syntactically data independent. $\diamond$

Summary

In this chapter we defined data-independent and data-dependent constructors, and for each constructor in our language proved that it was data independent or data dependent. We gave a definition of syntactic data independence based on whether a formula uses only data-independent constructors. Using an induction over language syntax we proved that formulas which satisfy the syntactic definition of data independence also satisfy the semantic definition. We showed that there are contrived instances where the converse does not hold. Using the syntactic definition one can automatically check a formula to see whether using the small model theorem is sound.

Chapter 5

Thresholds

In this chapter we calculate thresholds for the scope of data independent type variables and prove that analyses at such scopes are complete.

Recall that in Chapter 3 we gave a semantic definition of data independence. If one wishes to evaluate whether a data independent formula is true under some set map and binding, one can instead evaluate the formula under a quotient set map and quotient binding. These quotient set maps and bindings are obtained by choosing a data relation under which the original binding is closed.

Our first task in this chapter therefore will be to explain how to choose a data relation for a given binding. We will choose a maximum data relation: it will be the largest choice under the subset order, for which the binding is closed. Such a data relation induces a minimum number of equivalence classes on the carrier set of the data independent type variable.

Our second task will be to consider, for any binding, the number of equivalence classes induced by our maximum data relation. In many cases we can derive an upper bound on the number of equivalence classes, by consideration of the types of the variables in the binding.

When it comes to checking the consistency/validity of a data independent formula, this upper bound will do as a threshold scope on the data independent type variable. This is because, if there exists any set map and binding which makes the formula true/false, then one can take the appropriate quotient and find a set map and binding within the threshold scope that also makes the formula true/false.

5.1 Maximum data relations

As a stepping stone towards choosing a data relation which is maximum for a particular binding, we first define a data relation which is maximum for a particular value. Our maximum data relation for a particular binding will be the intersection of the maximum data relations for each value bound in the binding.

Definition 5.1 Maximum data relation for a value

Let $X : \mathit{TypeVar}$, $V : \mathit{Value}, \delta : \mathit{SetMap}$. Suppose $X \in \operatorname{dom}\delta$. We define the data relation induced by V for the single data independent type variable X, written $\sim_{(X,V)}$, by:

$$
\begin{array}{l}
x \sim_{(X,V)} y \Leftrightarrow \\
\quad x = y \lor \\
\quad (x \in \delta(X) \land y \in \delta(X) \land \\
\quad (\forall\, xs, ys : \operatorname{seq} \mathit{Atom} \bullet xs \frown \langle x \rangle \frown ys \in V \Leftrightarrow xs \frown \langle y \rangle \frown ys \in V))
\end{array}
$$

$\diamond$

This definition sets two elements in the carrier set of the data independent type to be equivalent for a value if swapping an occurrence of one for the other in any sequence of atoms does not change whether that sequence belongs to the value. It is helpful to think of the two atoms being equivalent from the perspective of the value.

Example 5.2

Suppose $\Upsilon = \{\texttt{Professor}, \texttt{Student}\}$ and $\delta : \mathit{SetMap}(\Upsilon)$. Suppose:

$$
\begin{array}{l}
\delta(\texttt{Professor}) = \{x_1, x_2, x_3, x_4, x_5\} \\
\delta(\texttt{Student}) = \{y_1, y_2\}
\end{array}
$$

Define:

$$
\begin{array}{l}
V = \{\langle y_1, x_1 \rangle, \\
\qquad \langle y_2, x_2 \rangle, \\
\qquad \langle y_2, x_3 \rangle\}
\end{array}
$$

Then $x_2 \sim_{(\texttt{Professor},V)} x_3$, because swapping an occurrence of x_2 for x_3 in a sequence in V yields a sequence in V and swapping x_2 for x_3 in a sequence not in V yields a sequence not in V. Similarly $x_4 \sim_{(\texttt{Professor},V)} x_5$. No other pair of elements in the carrier set of `Professor` are equivalent. $\diamond$

We now give a series of definitions and lemmas culminating in Lemma 5.6 which explains why we have called our choice of data relation *maximum*. It establishes that our choice of data relation is one for which the value is closed, and that it is greater than any other data relation for which the value is closed.

Definition 5.3 Equivalent sequences

An equivalence relation on atoms, $\sim$, induces an equivalence relation on sequences of atoms given by:

$$xs \sim ys \Leftrightarrow \#(xs) = \#(ys) \land (\forall\, i : 1 \mathinner{..} \#(xs) \bullet xs(i) \sim ys(i))$$

$\diamond$

Definition 5.4 Alternative definition of closed

Let V be a value and $\sim$ an equivalence relation on atoms. We define:

$$Closed^2_\sim(V) \Leftrightarrow \\ (\forall\, xs : \operatorname{seq} Atom;\ ys : \operatorname{seq} Atom \mid xs \sim ys \bullet xs \in V \Rightarrow ys \in V)$$

$\diamond$

Theorem 5.5 Both definitions of closed are equivalent
Let V be a value and $\sim$ an equivalence relation on atoms. Then:

$$Closed_\sim(V) \Leftrightarrow Closed^2_\sim(V)$$

$\diamond$

Proof See Theorem A.7 $\square$

Lemma 5.6
Let $X : TypeVar, V : Value, \delta : SetMap$. Suppose $X \in \operatorname{dom}\delta$. Let $\sim : DataRel_X(\delta)$. Then:

$$\sim \subseteq \sim_{(X,V)} \Leftrightarrow Closed_\sim(V)$$

$\diamond$

Proof Note $\sim$ induces a relation on $\operatorname{seq} Atom$ (as defined in Definition 5.3).
We first prove the forward implication. Suppose $\sim \subseteq \sim_{(X,V)}$. Let $xs : V, ys : \operatorname{seq} Atom$ and suppose $xs \sim ys$. For any $i : 1 \mathinner{..} \#(xs)$ define:

$$zs_i(j) = \textbf{if } j \geq i \textbf{ then } xs(j) \textbf{ else } ys(j)$$

Let $i : 1 \mathinner{..} \#(xs) - 1$. Then the following hold:

$$zs_i = (\lambda\, k : 1 \mathinner{..} i-1 \bullet ys(i)) \frown \langle xs(i) \rangle \frown (\lambda\, k : 1 \mathinner{..} \#(xs) - i \bullet xs(i+k))$$
$$zs_{i+1} = (\lambda\, k : 1 \mathinner{..} i-1 \bullet ys(i)) \frown \langle ys(i) \rangle \frown (\lambda\, k : 1 \mathinner{..} \#(xs) - i \bullet xs(i+k))$$
$$xs(i) \sim ys(i)$$
$$xs(i) \sim_{(X,V)} ys(i)$$

Thus:

$$zs_i \in V \Rightarrow zs_{i+1} \in V$$

And $xs = zs_1 \in V$. So by induction:

$$ys = zs_{\#xs} \in V$$

Thus using Definition 5.4 and Theorem 5.5 we have established:

$$Closed_\sim(V)$$

This completes the proof of the forward implication.

For the backward implication suppose $Closed_{\sim}(V)$. Let $x, y : Atom$. We split the remainder of the proof into two cases.

Case $x \notin \delta(X) \lor y \notin \delta(Y)$.

$$
\begin{array}{ll}
& x \sim y \\
\Rightarrow & [3.28] \\
& x = y \\
\Rightarrow & [5.1] \\
& x \sim_{(X,V)} y
\end{array}
$$

Case $x \in \delta(X) \land y \in \delta(Y)$. Let $xs, ys : \operatorname{seq} Atom$.

$$
\begin{array}{ll}
& x \sim y \\
\Rightarrow & [5.3] \\
& xs \frown \langle x \rangle \frown ys \sim xs \frown \langle y \rangle \frown ys \\
\Rightarrow & [5.4,\ 5.5] \\
& (xs \frown \langle x \rangle \frown ys \in V \Leftrightarrow xs \frown \langle y \rangle \frown ys \in V) \\
\Rightarrow & [5.1] \\
& x \sim_{(X,V)} y
\end{array}
$$

This completes the proof of the backward implication. □

Now we define the maximum data relation for a particular binding.

Definition 5.7 Maximum data relation for a binding

Let $X : TypeVar$. Let (Υ, Γ) be a schema. Let $\delta : SetMap$ and suppose $\operatorname{dom}\delta = \Upsilon$. Let $\eta : Binding(\Upsilon, \Gamma, \delta)$. We define the data relation induced by η for the single data independent type variable X, written $\sim_{(X,\eta)}$, by:

$$\sim_{(X,\eta)} = \bigcap \{ var : \operatorname{dom}\eta \bullet \sim_{(X,\eta(var))} \}$$

◇

The following lemma explains why our choice is the maximum. It establishes that this data relation is one for which the binding is closed, and that it is greater than any other data relation for which the binding is closed.

Lemma 5.8

Let (Υ, Γ) be a schema. Let $X : \Upsilon$. Let $\delta : SetMap(\Upsilon)$. Let $\eta : Binding(\Upsilon, \Gamma, \delta)$. Let $\sim : DataRel_X(\delta)$. Then:

$$\sim \subseteq \sim_{(X,\eta)} \Leftrightarrow Closed_{\sim}(\eta)$$

◇

Proof

$$
\begin{array}{ll}
& \sim \subseteq \sim_{(X,\eta)} \\
\Leftrightarrow & [5.7] \\
& (\forall\, var : \eta \bullet \sim \subseteq \sim_{(X,\eta(var))}) \\
\Leftrightarrow & [5.6] \\
& (\forall\, var : \eta \bullet \mathit{Closed}_{\sim}(\eta(var))) \\
\Leftrightarrow & [3.23] \\
& \mathit{Closed}_{\sim}(\eta)
\end{array}
$$

□

5.2 Folds and equivalence relations

In this section we give some definitions relating to folds and equivalence relations. Some standard results are also stated without proof. These definitions and lemmas will be used in the remaining part of this chapter.

Definition 5.9 Sequence Fold

We define a curried function *SeqFold*. Its first argument is binary operator and its second a sequence of operands. The result of applying *SeqFold* to its arguments, is an operand.

Let *Operand* be a set and $\otimes : \mathit{Operand} \times \mathit{Operand} \rightarrow \mathit{Operand}$ be an infix binary operator. We now define *SeqFold* inductively over the second argument:

$$
\begin{array}{l}
\mathit{SeqFold}(\otimes)(\langle x \rangle) = x \\
\mathit{SeqFold}(\otimes)(\langle x \rangle \frown xs) = x \otimes \mathit{SeqFold}(\otimes)(xs)
\end{array}
$$

◇

Definition 5.10 Range Fold

We now define a curried function *RanFold*. The first argument is a binary operator which is commutative and associative. The second argument is a function which has a finite, non-empty domain. The range of this function is a set of operands. The result of applying *RanFold* to its arguments is an operand.

Let *Operand* be a set and $\otimes : \mathit{Operand} \times \mathit{Operand} \rightarrow \mathit{Operand}$ be an infix binary operator, which is commutative and associative. Let *Index* be an arbitrary set and let $\mathit{Fun} : \mathit{Index} \rightarrow \mathit{Operand}$. Suppose $\mathrm{dom}\, \mathit{Fun}$ is non-empty and finite. We define $\mathit{RanFold}(\otimes)(\mathit{Fun})$ as follows:

Choose a bijection $\beta : (1 \,..\, \#(\mathrm{ran}\, \mathit{Fun})) \rightarrowtail\!\!\!\!\!\rightarrow (\mathrm{ran}\, \mathit{Fun})$. Then:

$$
\mathit{RanFold}(\otimes)(\mathit{Fun}) = \mathit{SeqFold}(\otimes)(\beta \fatsemi \mathit{Fun})
$$

The result is the same whatever the choice of the bijection, because the operator is commutative and associative. ◇

Definition 5.11 Replicated product
We define Π by:

$$\Pi = \mathit{RanFold}(*)$$

$\diamond$

Let Ω be a set.

Definition 5.12 Transitive, Symmetric and Reflexive Relations
Let $\sim : \Omega \leftrightarrow \Omega$.

$$\mathit{Transitive}(\sim) \Leftrightarrow (\forall\, x : \Omega;\ y : \Omega;\ z : \Omega \bullet x \sim y \land y \sim z \Rightarrow x \sim z)$$

$$\mathit{Symmetric}(\sim) \Leftrightarrow (\forall\, x : \Omega;\ y : \Omega;\ z : \Omega \bullet x \sim y \Rightarrow y \sim x)$$

$$\mathit{Reflexive}(\sim) \Leftrightarrow (\forall\, x : \Omega \bullet x \sim x)$$

$\diamond$

Definition 5.13 Equivalence Relations

$$\mathit{EqRel}_\Omega = \{\sim : (\Omega \leftrightarrow \Omega) \mid \mathit{Transitive}(\sim) \land \mathit{Symmetric}(\sim) \land \mathit{Reflexive}(\sim)\}$$

$\diamond$

Definition 5.14 Equivalence Classes
Let $\sim$: EqRel_Ω. Let $x : \Omega$.

$$[x]_\sim = \{y : \Omega \mid x \sim y\}$$

$\diamond$

Definition 5.15 Partitions

$$\mathit{Partition}_\Omega = \{p : \mathbb{P}(\mathbb{P}\,\Omega) \mid \bigcup p = \Omega \land (\forall\, b, c : p \bullet b = c \lor b \cap c = \emptyset)\}$$

$\diamond$

Definition 5.16 Combination of partitions
Let $p, q : \mathit{Partition}_\Omega$.

$$p \otimes q = \{b : p;\ c : q \bullet b \cap c\} \setminus \{\emptyset\}$$

$\diamond$

Lemma 5.17

Let $p, q : Partition_\Omega$. Then:

$p \otimes q \in Partition_\Omega$

$\diamond$

Definition 5.18 Partition induced by an equivalence relation
Let $\sim : EqRel_\Omega$.

$\Omega/\sim = \{x : \Omega \bullet [x]_\sim\}$

$\diamond$

Lemma 5.19
Let $\sim : EqRel_\Omega$. Then:

$\Omega/\sim \in Partition_\Omega$

$\diamond$

Lemma 5.20
Let $\sim_1, \sim_2 : EqRel_\Omega$. Then:

$\sim_1 \cap \sim_2 \in EqRel_\Omega$
$\Omega/(\sim_1 \cap \sim_2) = (\Omega/\sim_1) \otimes (\Omega/\sim_2)$

$\diamond$

Lemma 5.21
Let $Rels : \mathbb{P}(EqRel_\Omega)$ Then:

$\bigcap Rels \in EqRel_\Omega$
$\Omega/(\bigcap Rels) = RanFold(\otimes)\{\sim : Rels \bullet \sim \mapsto (\Omega/\sim)\}$

$\diamond$

Lemma 5.22
Let $\sim_1, \sim_2 : EqRel_\Omega$ and suppose $\sim_1 \subseteq \sim_2$. Then:

$\#(\Omega/\sim_1) \geq \#(\Omega/\sim_2)$

$\diamond$

5.3 Calculation of thresholds

To calculate a threshold on the size of a data independent type variable in a formula we need to find a bound[1] over all bindings on the number of equivalence classes the maximum data relation of the binding induces on the carrier set of the data independent type variable.

We first establish that such a bound can be obtained by finding a bound on the number of equivalence classes induced by the maximum data relation for each of the values in the binding. We then use the type of these values to establish a bound on the number of equivalence classes each induces.

Lemma 5.23

Let Ω be a set and let $\sim_1, \sim_2 : EqRel_\Omega$. Then:

$$\#(\Omega/(\sim_1 \cap \sim_2)) \leq \#(\Omega/\sim_1) * \#(\Omega/\sim_2)$$

$\diamond$

Proof

$$\begin{array}{ll} & \#(\Omega/(\sim_1 \cap \sim_2)) \\ = & [5.20] \\ & \#((\Omega/\sim_1) \otimes (\Omega/\sim_2)) \\ = & [5.16] \\ & \#(\{b : \Omega/\sim_1;\ c : \Omega/\sim_2 \bullet b \cap c\} \setminus \{\emptyset\}) \\ \leq & \\ & \#(\Omega/\sim_1) * \#(\Omega/\sim_1) \end{array}$$

$\square$

The above lemma shows that the number of equivalence classes induced by the intersection of two equivalence relations is no greater than the product of the number of equivalence classes induced by each one. This result generalizes from two to an arbitrary finite number of equivalence relations. This is because the intersection operation on equivalence relations is commutative and associative and the same can be said for the product operation on natural numbers.

Lemma 5.24

Let Ω be a set. Let $Rels : \mathbb{P}\, EqRel_\Omega$ and suppose $Rels$ is finite and non-empty. Then:

$$\#(\Omega/\bigcap Rels) \leq \Pi\{\sim : Rels \bullet \sim \mapsto \#(\Omega/\sim)\}$$

where Π, a replicated product operator, was defined in Definition 5.11.

[1] Do not confuse the noun 'bound' meaning limit and the past participle 'bound' meaning pinned down, since both senses are relevant to our present discussion. Which one we mean will be clear from the grammar!

$\diamond$

Now the maximum data relation for a binding is just the intersection of the maximum data relations for the each value in the binding (see Definition 5.7). So this lemma tells us to multiply together the bounds on the number of equivalence classes induced by each value, to get a bound on the number of equivalence classes induced by the binding.

The maximum data relations we have so far defined are relations on the whole of *Atom*, and atoms which are not in the carrier set of the data independent type variable relate to themselves only. While this was useful in the earlier part of the thesis, it would be cumbersome for this part. We therefore define the restriction of these relations to the carrier set of the data independent type to give a better notation and allow us to apply the above lemma more readily.

Definition 5.25 Restricted maximum data relations

Let $X : \mathit{TypeVar}$, $V : \mathit{Value}$. Let (Υ, Γ) be a schema. Let $\delta : \mathit{SetMap}$ and suppose $\mathrm{dom}\,\delta = \Upsilon$. Let $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$.

We define:

$$\approx_{(X,V)} = \sim_{(X,V)} \cap (\delta(X) \times \delta(X))$$
$$\approx_{(X,\eta)} = \sim_{(X,\eta)} \cap (\delta(X) \times \delta(X))$$

$\diamond$

The following lemma reflects the change of notation.

Lemma 5.26

Let $X : \mathit{TypeVar}$, $V : \mathit{Value}$. Let (Υ, Γ) be a schema. Let $\delta : \mathit{SetMap}$ and suppose $\mathrm{dom}\,\delta = \Upsilon$. Let $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$. Then:

$$(\delta/\sim_{(X,V)})(X) = \delta(X)/\approx_{(X,V)}$$
$$(\delta/\sim_{(X,\eta)})(X) = \delta(X)/\approx_{(X,\eta)}$$

$\diamond$

Proof

$$\begin{array}{ll}
& (\delta/\sim_{(X,V)})(X) \\
= & [3.19] \\
& \{x : \delta(X) \bullet [x]_{\sim_{(X,V)}}\} \\
= & \\
& \{x : \delta(X) \bullet \{y : \mathit{Atom} \mid x \sim_{(X,V)} y\}\} \\
= & \\
& \{x : \delta(X) \bullet \{y : \delta(X) \mid x \sim_{(X,V)} y\}\} \\
= & \\
& \{x : \delta(X) \bullet \{y : \delta(X) \mid x \approx_{(X,V)} y\}\}
\end{array}$$

$$
\begin{aligned}
&= \\
&\{x : \delta(X) \bullet [x]_{\approx_{(X,V)}}\} \\
&= \\
&\delta(X)/\approx_{(X,V)}
\end{aligned}
\qquad [5.18]
$$

This completes the proof of the first consequent of the lemma. The second follows similarly. □

5.3.1 Cardinality of types

We will soon determine the number of equivalence classes arising from (the values of) general type expressions. This determination makes use of bounds on: $\#([T]_\delta)$ and $\#([\![T]\!]_\delta)$ for any subexpressions T which do not use data independent type variables. Such bounds exist when a finite scope is given for the type variables which are not data independent i.e. every type variable used in T. In this subsection we define such bounds.

Definition 5.27 Size and Cardinality

Let $\Upsilon : \mathbb{P}\, TypeVar$. Let $\Theta : Scope(\Upsilon)$. We define $Size_\Theta : TypeExp \nrightarrow \mathbb{N}^\infty$ and $Card_\Theta : TypeExp \nrightarrow \mathbb{N}^\infty$ as follows.

Let $T : TypeExp$ and suppose $Used(T) \subseteq \Upsilon$. We define:

$$Size_\Theta(T) = max\{\delta : SetMap(\Upsilon) \mid \# \circ \delta \leq \Theta \bullet \#([T]_\delta)\}$$

$$Card_\Theta(T) = max\{\delta : SetMap(\Upsilon) \mid \# \circ \delta \leq \Theta \bullet \#([\![T]\!]_\delta)\}$$

◇

Now although $\#([T]_\delta)$ depends on δ, it depends on the size of the carrier sets δ assigns to type variables, not the carrier sets themselves. So for these purposes two set maps can be considered isomorphic if they map type variables to carrier sets with the same sizes.

When Θ is finite (on $Used(T)$) then the number of possible set maps within the scope Θ is finite up to isomorphism. Thus $Size_\Theta(T)$ is the maximum over a finite set of values, each of which can be easily calculated. Hence one can automate the calculation of $Size_\Theta(T)$ with a brute-force approach. The same can be said for $Card_\Theta(T)$.

An alternative approach would be to define $Size_\Theta(T)$ recursively over the structure of T such that $Size_\Theta(T)$ has the property of being a bound on $\#([T]_\delta)$, which we might hope to prove inductively. This is indeed possible for $Size_\Theta(T)$ and would give a more efficient method of calculation. However this approach does not seem to work well for $Card_\Theta(T)$, because it is not possible to calculate $Card_\Theta(T)$, from the value of $Card_\Theta$ applied to the subexpressions of T in general unless tolerating very loose bounds. One way round this would be to introduce other functions, in effect strengthening the inductive hypothesis of the associated proof that $Card_\Theta(T)$ is a bound, however this soon becomes quite complex.

The calculation of bounds on type expressions which do not include data independent type variables, is considered not to be at the core of this thesis and therefore we have adopted the former, simpler approach. The approach could be refined at some point in the future.

5.3.2 Types without multiplicity keywords

In this section we calculate, for each fixed arity type, the maximum number of equivalence classes a value of the type induces on the carrier set of the data independent type variable, via its maximum data relation. The following function performs this calculation.

Definition 5.28 Regions

Let Υ be a set of type variables. Let $\Theta : \mathit{Scope}(\Upsilon)$. Let $X : \Upsilon$. We define $\mathit{Regions}_{X,\Theta} : \mathit{FixedArityType} \to \mathbb{N}^\infty$ as follows:

$$\mathit{Regions}_{X,\Theta}(Y) = \textbf{if } Y = X \textbf{ then } 2 \textbf{ else } 1$$
$$\mathit{Regions}_{X,\Theta}(P \texttt{ + } Q) = \mathit{Regions}_{X,\Theta}(P) * \mathit{Regions}_{X,\Theta}(Q)$$

where $Y : \mathit{TypeVar}$ and $P, Q : \mathit{FixedArityType}$.

The value of $\mathit{Regions}_{X,\Theta}(P \texttt{ -> } Q)$ is given in the following table:

	$\mathit{Regions}_{X,\Theta}(Q) = 1$	$\mathit{Regions}_{X,\Theta}(Q) > 1$
$\mathit{Regions}_{X,\Theta}(P) = 1$	1	$\mathit{Regions}_{X,\Theta}(Q)^{\mathit{Size}_\Theta(P)}$
$\mathit{Regions}_{X,\Theta}(P) > 1$	$\mathit{Regions}_{X,\Theta}(P)^{\mathit{Size}_\Theta(Q)}$	∞

$\diamond$

The following theorem tell us, $\mathit{Regions}_{X,\Theta}(P)$ gives a bound on the number of equivalence classes on the carrier set of X induced by (the maximum data relation induced by) any variable of fixed arity type P.

Theorem 5.29 *Regions* gives a valid bound.

Let Υ be a set of type variables. Let $\Theta : \mathit{Scope}(\Upsilon)$. Let $\delta : \mathit{SetMap}(\Upsilon)$ and suppose $\# \circ \delta \leq \Theta$. Let $P : \mathit{FixedArityType}$ and suppose $\mathit{Used}(P) \subseteq \Upsilon$. Let $V : [\![P]\!]_\delta$. Then:

$$\#(\delta(X)/\approx_{(X,V)}) \leq \mathit{Regions}_{X,\Theta}(P)$$

$\diamond$

Proof We proceed by induction over the structure of P.

Case Y.

Let $x_1, x_2 : \delta(X)$.

Sub-case $Y \neq X$.

We know $x_1 \notin \delta(Y) \wedge x_2 \notin \delta(Y)$ and it follows:

$$V \in [\![Y]\!]_\delta$$
$$\Rightarrow$$

$$
\begin{array}{l}
V \subseteq \{y : \delta(Y) \bullet \langle y \rangle\} \\
\Rightarrow \\
(\forall\, ys : V \bullet \operatorname{ran} ys \subseteq \delta(Y))
\end{array}
$$

Now let $xs, ys : \operatorname{seq} Atom$. Then:

$$
\begin{array}{l}
\neg\ \operatorname{ran}(xs \frown \langle x_1 \rangle \frown ys) \subseteq \delta(Y) \\
\Rightarrow \\
xs \frown \langle x_1 \rangle \frown ys \notin V
\end{array}
$$

Similarly $xs \frown \langle x_2 \rangle \frown ys \notin V$. Thus:

$$
xs \frown \langle x_1 \rangle \frown ys \in V \Leftrightarrow xs \frown \langle x_2 \rangle \frown ys \in V
$$

We have now shown $x_1 \sim_{(X,V)} x_2$ for any $x_1, x_2 : \delta(X)$. Therefore:

$$
\begin{array}{l}
\delta(X)/\approx_{(X,V)} = \{\delta(X)\} \\
\Rightarrow \\
\#(\delta(X)/\approx_{(X,V)}) = 1 \leq \mathit{Regions}_{X,\Theta}(Y)
\end{array}
$$

Sub-case $Y = X$.

$$
\forall\, ys : V \bullet \#(ys) = 1
$$

Let $xs, ys : \operatorname{seq} Atom$.

$$
\begin{array}{l}
xs \frown \langle x_1 \rangle \frown ys \in V \\
\Rightarrow \\
\#(xs \frown \langle x_1 \rangle \frown ys) = 1 \\
\Rightarrow \\
xs = \langle\rangle \wedge ys = \langle\rangle
\end{array}
$$

Thus:

$$
\begin{array}{l}
x_1 \sim_{(X,V)} x_2 \\
\Leftrightarrow \\
(\forall\, xs, ys : \operatorname{seq} Atom \bullet xs \frown \langle x_1 \rangle \frown ys \in V \Leftrightarrow xs \frown \langle x_1 \rangle \frown ys \in V) \\
\Leftrightarrow \\
(\langle x_1 \rangle \in V \Leftrightarrow \langle x_2 \rangle \in V)
\end{array}
$$

Hence:

$$
\begin{array}{l}
\delta(X)/\approx_{(X,V)} \\
=
\end{array}
$$

$$
\begin{array}{l}
\{x : \delta(X) \bullet [x]_{\approx_{(X,V)}}\} \\
= \\
\{x : \delta(X) \bullet \{y : \delta(X) \mid x \approx_{(X,V)} y\}\} \\
= \\
\{x : \delta(X) \bullet \{y : \delta(X) \mid \langle x \rangle \in V \Leftrightarrow \langle y \rangle \in V\}\} \\
= \\
\{x : \delta(X) \mid \langle x \rangle \in V \bullet \{y : \delta(X) \mid \langle x \rangle \in V \Leftrightarrow \langle y \rangle \in V\}\} \\
\cup \\
\{x : \delta(X) \mid \langle x \rangle \notin V \bullet \{y : \delta(X) \mid \langle x \rangle \in V \Leftrightarrow \langle y \rangle \in V\}\} \\
= \\
\{x : \delta(X) \mid \langle x \rangle \in V \bullet \{y : \delta(X) \mid \langle y \rangle \in V\}\} \\
\cup \\
\{x : \delta(X) \mid \langle x \rangle \notin V \bullet \{y : \delta(X) \mid \langle y \rangle \notin V\}\} \\
\subseteq \\
\{\{y : \delta(X) \mid \langle y \rangle \in V\}, \{y : \delta(X) \mid \langle y \rangle \notin V\}\}
\end{array}
$$

Thus:

$$
\begin{array}{l}
\#(\delta(X)/\approx_{(X,V)}) \\
\leq \\
\#(\{\{y : \delta(X) \mid \langle y \rangle \in V\}, \{y : \delta(X) \mid \langle y \rangle \notin V\}\}) \\
\leq \\
2 = \mathit{Regions}_{X,\Theta}(X)
\end{array}
$$

Case $P + Q$.

$$
\begin{array}{l}
V \in [\![P + Q]\!]_\delta \\
\Rightarrow \\
V \subseteq [P + Q]_\delta = [P]_\delta \cup [Q]_\delta
\end{array}
$$

Let $V_1 = V \cap [P]_\delta$ and $V_2 = V \cap [Q]_\delta$. Then: $V = V_1 \cup V_2$. We will now show:

$$\approx_{(X,V_1)} \cap \approx_{(X,V_2)} \subseteq \approx_{(X,V)}$$

Let $x_1, x_2 : \delta(X)$ and suppose $x_1 \approx_{(X,V_1)} x_2 \wedge x_1 \approx_{(X,V_2)} x_2$. Let $xs, ys : \mathrm{seq}\, \mathit{Atom}$.

$$
\begin{array}{l}
xs \frown \langle x_1 \rangle \frown ys \in V \\
\Leftrightarrow \\
xs \frown \langle x_1 \rangle \frown ys \in V_1 \vee xs \frown \langle x_1 \rangle \frown ys \in V_2 \\
\Leftrightarrow
\end{array}
$$

$$xs \frown \langle x_2 \rangle \frown ys \in V_1 \lor xs \frown \langle x_2 \rangle \frown ys \in V_2$$
$$\Leftrightarrow$$
$$xs \frown \langle x_2 \rangle \frown ys \in V$$

This shows:

$$\approx_{(X,V_1)} \cap \approx_{(X,V_2)} \subseteq \approx_{(X,V)}$$

Thus:

$$\begin{array}{ll} & \#(\delta(X)/\approx_{(X,V)}) \\ \leq & \quad [5.22] \\ & \#(\delta(X)/(\approx_{(X,V_1)} \cap \approx_{(X,V_2)})) \\ \leq & \quad [5.23] \\ & \#(\delta(X)/\approx_{(X,V_1)}) * \#(\delta(X)/\approx_{(X,V_2)}) \\ \leq & \quad [\text{ind. hyp}] \\ & \mathit{Regions}_{X,\Theta}(P) * \mathit{Regions}_{X,\Theta}(Q) = \mathit{Regions}_{X,\Theta}(P \texttt{ + } Q) \end{array}$$

Case P -> Q.
Sub-case $\mathit{Regions}_{X,\Theta}(P) = 1$.
Let $V : [\![P \texttt{ -> } Q]\!]_\delta$. Let:

$$W = \{xs : [P]_\delta;\ ys : [Q]_\delta \mid xs \frown ys \in V \bullet xs\}$$

Then:

$$W \subseteq [P]_\delta$$
$$\Rightarrow$$
$$\#(W) \leq \mathit{Size}_\Theta(P)$$

For each $ws : W$ define:

$$V_{ws} = \{ys : [Q]_\delta \mid ws \frown ys \in V\}$$

and note for future reference $V_{ws} \subseteq [Q]_\delta$ i.e. $V_{ws} \in [\![Q]\!]_\delta$.
We claim:

$$\bigcap\{ws : W \bullet \approx_{(X,V_{ws})}\} = \approx_{(X,V)}$$

and will now show inclusion in both directions.
Let $x_1, x_2 : \delta(X)$ and suppose:

$$(x_1, x_2) \in \bigcap\{ws : W \bullet \approx_{(X,V_{ws})}\}$$

Let $xs, ys : \operatorname{seq} Atom$ and suppose:

$$xs \frown \langle x_1 \rangle \frown ys \in V$$

Using Lemma 5.30, we may write $xs \frown \langle x_1 \rangle \frown ys = ws \frown vs \frown \langle x_1 \rangle \frown ys$ where $ws \in W$ and $vs \frown \langle x_1 \rangle \frown ys \in V_{ws}$. Then:

$$\begin{array}{l} x_1 \approx_{(X, V_{ws})} x_2 \\ \Rightarrow \\ vs \frown \langle x_2 \rangle \frown ys \in V_{ws} \\ \Rightarrow \\ ws \frown vs \frown \langle x_2 \rangle \frown ys \in V \\ \Rightarrow \\ xs \frown \langle x_2 \rangle \frown ys \in V \end{array}$$

Hence:

$$\forall\, xs, ys : \operatorname{seq} Atom \bullet (xs \frown \langle x_1 \rangle \frown ys \in V \Rightarrow xs \frown \langle x_2 \rangle \frown ys \in V)$$

And similarly:

$$\forall\, xs, ys : \operatorname{seq} Atom \bullet (xs \frown \langle x_1 \rangle \frown ys \in V \Rightarrow xs \frown \langle x_2 \rangle \frown ys \in V)$$

Thus $x_1 \approx_{(X, V)} x_2$ completing the proof of the forwards inclusion.

Let $x_1, x_2 : \delta(X)$ and suppose:

$$x_1 \approx_{(X, V)} x_2$$

Let $ws : W$. Let $xs, ys : \operatorname{seq} Atom$ and suppose:

$$xs \frown \langle x_1 \rangle \frown ys \in V_{ws}$$

If follows:

$$\begin{array}{l} ws \frown xs \frown \langle x_1 \rangle \frown ys \in V \\ \Rightarrow \\ ws \frown xs \frown \langle x_2 \rangle \frown ys \in V \\ \Rightarrow \\ xs \frown \langle x_1 \rangle \frown ys \in V_{ws} \end{array}$$

Hence:

$$\forall\, xs, ys : \operatorname{seq} Atom \bullet (xs \frown \langle x_1 \rangle \frown ys \in V_{ws} \Rightarrow xs \frown \langle x_2 \rangle \frown ys \in V_{ws})$$

And similarly:

$$\forall\, xs, ys : \operatorname{seq} Atom \bullet (xs \frown \langle x_2 \rangle \frown ys \in V_{ws} \Rightarrow xs \frown \langle x_1 \rangle \frown ys \in V_{ws})$$

Thus:

$$\forall\, ws : W \bullet (x_1, x_2) \in \approx_{(X, V_{ws})}$$

completing the proof of the backwards inclusion.

Now recall that for each $ws : W$, $V_{ws} \in [\![Q]\!]_\delta$. The inductive hypothesis gives:

$$\forall\, ws : W \bullet \#(\delta(X)/\approx_{(X, V_{ws})}) \leq Regions_{X,\Theta}(Q)$$

Thus:

$$\begin{array}{ll} & \#(\delta(X)/\approx_{(X,V)}) \\ & \leq \qquad [5.24] \\ & \Pi\{ws : W \bullet ws \mapsto \#(\delta(X)/\approx_{(X,V_{ws})})\} \\ & \leq \\ & \Pi\{ws : W \bullet ws \mapsto Regions_{X,\Theta}(Q)\} \\ & \leq \\ & (Regions_{X,\Theta}(Q))^{Size_\Theta(Q)} \\ & = \\ & Regions_{X,\Theta}(P \text{ -> } Q) \end{array}$$

Sub-case $Regions_{X,\Theta}(Q) = 1$.
Similarly.
Sub-case $Regions_{X,\Theta}(P) \neq 1$ and $Regions_{X,\Theta}(Q) \neq 1$.
$Regions_{X,\Theta}(P \text{ -> } Q) = \infty$, so nothing to prove.

□

The above proof uses the following lemma, which tells us that if a type expression induces only one equivalence class on the carrier set of the data-independent type variable then the type expression does not use the data-independent type variable at all, and, furthermore, that any sequence in the semantics of the type expression does not contain any element of the carrier set of the data-independent type variable.

Lemma 5.30
Let $\Upsilon : \mathbb{P}\, TypeVar, X : \Upsilon, P : FixedArityType$. Let $\delta : SetMap(\Upsilon)$. Then:

$$\begin{array}{l} Regions_{X,\Theta}(P) = 1 \\ \Rightarrow \\ X \notin Used(P) \\ \Rightarrow \\ (\forall\, ys : [P]_\delta \bullet \operatorname{ran} ys \cap \delta(X) = \emptyset) \end{array}$$

$\diamond$

Proof The proof is an easy induction over the structure of P. $\square$

Example 5.31

We will now give an example to show an application of Theorem 5.29. Let:

$$\begin{aligned}
\Upsilon &= \{\texttt{Professor}, \texttt{Student}, \texttt{Subject}\} \\
\Theta &= \{\texttt{Professor} \mapsto 5, \\
&\qquad \texttt{Student} \mapsto 1, \\
&\qquad \texttt{Subject} \mapsto 2\} \\
\delta &= \{\texttt{Professor} \mapsto \{x_1, x_2, x_3, x_4, x_5\}, \\
&\qquad \texttt{Student} \mapsto \{y_1\}, \\
&\qquad \texttt{Subject} \mapsto \{z_1, z_2\}\}
\end{aligned}$$

Theorem 5.29 tells us:

$$\begin{aligned}
&\forall\, V : [\![\texttt{Professor -> (Student -> Subject)}]\!]_\delta \bullet \\
&\quad \#(\delta(X)/{\approx_{(\texttt{Professor}, V)}}) \\
&\quad \leq \\
&\quad \mathit{Regions}_{\texttt{Professor},\Theta}(\texttt{Professor -> (Student -> Subject)})
\end{aligned}$$

We calculate $\mathit{Regions}_{\texttt{Professor},\Theta}(\texttt{Professor -> (Student -> Subject)})$ as follows:

$$\begin{aligned}
&\mathit{Regions}_{\texttt{Professor},\Theta}(\texttt{Student}) = \mathit{Regions}_{\texttt{Professor},\Theta}(\texttt{Subject}) = 1 \\
&\Rightarrow \\
&\mathit{Regions}_{\texttt{Professor},\Theta}(\texttt{Student -> Subject}) = 1 \\
&\Rightarrow \\
&\mathit{Regions}_{\texttt{Professor},\Theta}(\texttt{Professor -> (Student -> Subject)}) = \\
&\quad \mathit{Regions}_{\texttt{Professor},\Theta}(\texttt{Professor})^{\mathit{Size}_\Theta(\texttt{Student->Subject})} = \\
&\quad 2^{\mathit{Size}_\Theta(\texttt{Student->Subject})}
\end{aligned}$$

$$\begin{aligned}
&\mathit{Size}_\Theta(\texttt{Student -> Subject}) \\
&= \\
&\#([\texttt{Student -> Subject}]_\delta) \\
&= \\
&\#\{\langle y_1, z_1\rangle, \langle y_1, z_2\rangle\} \\
&= \\
&2
\end{aligned}$$

Hence $\mathit{Regions}_{\texttt{Professor},\Theta}(\texttt{Professor -> (Student -> Subject)}) = 2^2 = 4$. So, for

example, if

$$V = \{ \begin{array}{l} \langle x_1, y_1, z_1 \rangle, \\ \langle x_1, y_1, z_2 \rangle, \\ \langle x_2, y_1, z_1 \rangle, \\ \langle x_3, y_1, z_2 \rangle \} \end{array}$$

Then $V \in [\![\texttt{Professor -> (Student -> Subject)}]\!]_\delta$ and the theorem predicts $\#(\delta(\texttt{Professor})/\approx_{(\texttt{Professor}, V)}) \leq 4$ which is correct, since:

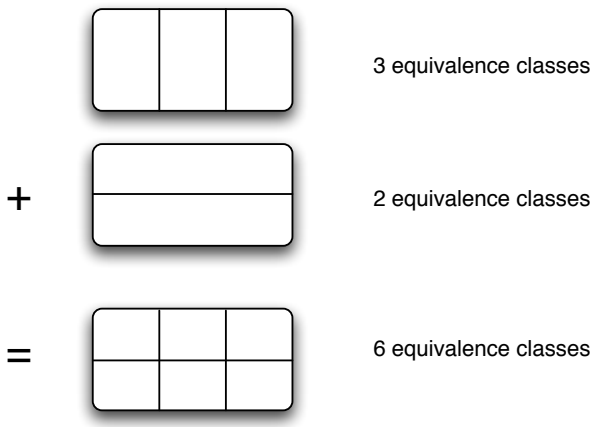
$$\delta(\texttt{Professor})/\approx_{(\texttt{Professor}, V)} = \{\{x_1\}, \{x_2\}, \{x_3\}, \{x_4, x_5\}\}$$

$\diamond$

5.3.3 Contributions

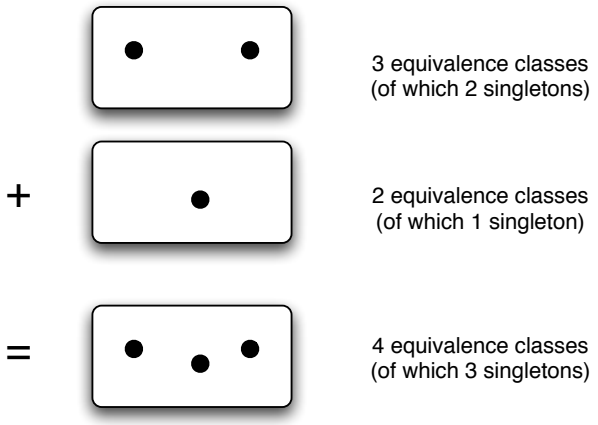
One can sometimes determine from the type of a value, not just the number of equivalence classes that will be induced by its maximum data relation, but that some of these classes will be singletons. This is the case when multiplicity keywords such as `one` and `lone` appear in the type (as will be seen in Section 5.3.4). Lower bounds can be obtained by using this extra information.

We make the point with a diagram, before continuing formally. Consider two equivalence relations on some set. The first induces three equivalence classes and the second two. Lemma 5.23 tells us that there are at most six equivalence classes induced by the intersection of these relations. We can picture these equivalence classes as regions of the plane as follows:



Suppose now we have extra information that two of the equivalence classes induced by the first equivalence relation are singletons, and one of the equivalence classes induced by the second equivalence relation is a singleton. We can

then determine that the intersection of the two equivalence relations induces at most four equivalence classes. We picture these singleton equivalence classes as points in the plane as follows:



Thus, although ultimately we are interested in calculating the number of equivalence classes for the whole binding, it is worth recording for the component values not just the number of equivalence classes but the number of these that are known to be singletons. The extra information can be used to obtain lower thresholds.

We now define a type *Contribution* as a pair of $\mathbb{N}^\infty$. The first records the number of equivalence classes known to be singletons, the second records the number of other equivalence classes.

Definition 5.32 Contributions

$$Contribution = \mathbb{N}^\infty \times \mathbb{N}^\infty$$

We will denote an element of *Contribution* by the letter C or D. $\diamond$

It is important that the result of combining contributions is a correct reflection of combining the partitions they represent, if we want to use combinations to calculate thresholds. We therefore give a semantics to contributions based on partitions.

Definition 5.33 Contribution semantics

We define the meaning of a contribution, C with respect to a set Ω as a subset of the partitions of Ω.

$$\begin{aligned}[\![C]\!]_\Omega = \{p : Partition_\Omega \mid \exists\, S : \mathbb{P}(\mathbb{P}\,\Omega);\ R : \mathbb{P}(\mathbb{P}\,\Omega) \bullet \\ (\forall\, b : S \bullet \#(b) = 1) \wedge \\ S \cap R = \emptyset \wedge S \cup R = p \wedge \\ \#(S) \leq first(C) \wedge \#(R) \leq second(C)\}\end{aligned}$$

$\diamond$

Thus the meaning of a contribution C is the set of partitions which are the union of disjoint sets of blocks S and R, where every block in S is a singleton and the sizes of S and R are at most the first and second components of C, respectively.

Definition 5.34 Combination operator

We now define an operator $* : \mathit{Contribution} \times \mathit{Contribution} \to \mathit{Contribution}$ as follows:

$$C * D = (\mathit{first}(C) + \mathit{first}(D), \mathit{second}(C) * \mathit{second}(D))$$

$\diamond$

The following lemma relates the combination of contributions with the combination of partitions they represent and thus allows us to use combinations in calculating thresholds.

Lemma 5.35 Combination of two contributions.

Let Ω be a set. Let $C, D : \mathit{Contribution}$. Let $p : [\![C]\!]_\Omega$ and $q : [\![D]\!]_\Omega$. Then:

$$p \otimes q \in [\![C * D]\!]_\Omega$$

N.B. $\otimes$ is defined in Definition 5.16. $\diamond$

Proof Choose $S_1, R_1, S_2, R_2 : \mathbb{P}(\mathbb{P}\,\Omega)$ such that:

$$p = S_1 \cup R_1 \wedge S_1 \cap R_1 = \emptyset$$
$$\forall\, b : S_1 \bullet \#(b) \leq 1$$
$$\#(S_1) \leq \mathit{first}(C) \wedge \#(R_1) \leq \mathit{second}(C)$$

$$q = S_2 \cup R_2 \wedge S_2 \cap R_2 = \emptyset$$
$$\forall\, b : S_2 \bullet \#(b) \leq 1$$
$$\#(S_2) \leq \mathit{first}(D) \wedge \#(R_2) \leq \mathit{second}(D)$$

Then let:

$$J = \{b_1 : S_1;\ b_2 : S_2 \bullet b_1 \cap b_2\} \setminus \{\emptyset\}$$
$$K = \{b_1 : S_1;\ b_2 : R_2 \bullet b_1 \cap b_2\} \setminus \{\emptyset\}$$
$$L = \{b_1 : R_1;\ b_2 : S_2 \bullet b_1 \cap b_2\} \setminus \{\emptyset\}$$
$$M = \{b_1 : R_1;\ b_2 : R_2 \bullet b_1 \cap b_2\} \setminus \{\emptyset\}$$

Then:

$$p_1 \otimes p_2$$
$$= \qquad [5.16]$$
$$\{b_1 : p_1;\ b_2 : p_2 \bullet b_1 \cap b_2\} \setminus \{\emptyset\}$$

$$=$$
$$(J \cup K \cup L) \cup M$$

Now:

$$(\forall\, b_1 : S_1 \bullet \#(b_1) \leq 1)$$
$$\Rightarrow$$
$$(\forall\, b_1 : S_2 \bullet \forall\, b_2 : R_2 \bullet b_1 \cap b_2 = b_1 \vee b_1 \cap b_2 = \emptyset)$$
$$\Rightarrow$$
$$K \subseteq S_1$$

Similarly $J \subseteq S_1$ and $L \subseteq S_2$. It follows $(J \cup K \cup L) \subseteq S_1 \cup S_2$. Thus:

$$\#(J \cup K \cup L) \leq \#(S_1 \cup S_2) \leq \mathit{first}(C) + \mathit{first}(D)$$
$$\forall\, b : (J \cup K \cup L) \bullet \#(b) \leq 1$$

Note also:

$$\#(M) \leq \#(R_1) * \#(R_2) \leq \mathit{second}(C) * \mathit{second}(D)$$

So to complete the proof it is only necessary to show:

$$(J \cup K \cup L) \cap M = \emptyset$$

So let $b : (J \cup K \cup L)$ and $c : M$. Choose $c_1 : R_1;\ c_2 : R_2$ in such a way that $c = c_1 \cap c_2$. Either $b \in S_1$ or $b \in S_2$.

Case $b \in S_1$.

$$b \in p_1 \wedge c_1 \in p_1$$
$$\Rightarrow \qquad [5.15]$$
$$b = c_1 \vee b \cap c_1 = \emptyset$$

Furthermore b and c_1 belong to disjoint sets: S_1 and R_1, so $b \neq c_1$ and hence $b \cap c_1 = \emptyset$. Thus:

$$b \cap c = b \cap c_1 \cap c_2 = \emptyset \cap c_2 = \emptyset$$
$$\Rightarrow$$
$$b \neq c$$

Case $b \in S_2$. Similarly.
Hence:

$$(J \cup K \cup L) \cap M = \emptyset$$

□

The above theorem generalizes from two partitions to an arbitrary finite

number of partitions, because the operators $\otimes$ and $*$ are associative and commutative.

Lemma 5.36 Combination of contributions.

Let Ω be a set. Let $ps : \mathbb{P}(Partition_\Omega)$ and suppose ps is non-empty and finite. Let $Cs : ps \to Contribution$ and suppose:

$$\forall\, p : ps \bullet p \in [\![Cs(p)]\!]_\Omega$$

Then:

$$RanFold(\otimes)(\mathrm{id}\, ps) \in [\![RanFold(*)(Cs)]\!]_\Omega$$

Note: *RanFold* is defined in 5.10. ◇

We will find it useful for the following subsection to introduce a power operation on contributions.

Definition 5.37 Power operator

Let $i : \mathbb{N}$ and suppose $i \geq 1$. Let $C : Contribution$. We define C^i by:

$$C^i = RanFold(*)(1 \mathinner{..} i \times \{C\})$$

Note that is follows immediately that:

$$C^i = (i * first(C), second(C)^i)$$

◇

5.3.4 Types with multiplicity keywords

In this section we determine the contribution arising from types which include multiplicity keywords.

We define a function which maps type expressions (see Definition 2.15) to contributions. This function is partial and we start by defining *Class A* and *Class B* type expressions, the union of which shall be the domain of this function.

Definition 5.38 Class A type expressions

$$ClassA = \{ T : TypeExp \mid (\exists\, m : Multi;\ P : FixedArityType \bullet T = m\ P) \}$$

◇

ClassA type expressions are those which consist of any multiplicity keyword, followed by any fixed arity type.

ClassB type expressions begin with `set` and contain up to one use of m `->` n where $m \neq$ `set` $\vee\ n \neq$ `set` . We may assume without loss of generality that such an occurrence of the construct m `->` n where $m \neq$ `set` $\vee\ n \neq$ `set` , applies to elements of *FixedArityType*. If this were not the case, the subexpressions to which the construct applies can be rewritten without changing their meaning

by replacing `set -> set` with `->`, thus becoming members of *FixedArityType*. (See also Convention 2.21).

Definition 5.39 Class B type expressions

$$\begin{aligned} \mathit{ClassB} = \{\, T : \mathit{TypeExp} \mid (\exists\, U : \mathit{RelType} \bullet \\ T = \texttt{set}\ U \land \mathit{LeafMultiProd}(U))\} \end{aligned}$$

where *LeafMultiProd* is a predicate over *RelType*, defined using pattern matching by:

$$\begin{aligned} &\mathit{LeafMultiProd}(P) \Leftrightarrow \mathit{true} \\ &\mathit{LeafMultiProd}(P\ m\texttt{->}n\ Q) \Leftrightarrow \mathit{true} \\ &\mathit{LeafMultiProd}(P\ \texttt{set->set}\ U) \Leftrightarrow \mathit{LeafMultiProd}(U) \\ &\mathit{LeafMultiProd}(T\ \texttt{set->set}\ Q) \Leftrightarrow \mathit{LeafMultiProd}(T) \\ &\mathit{LeafMultiProd}(P\ m\texttt{->}n\ U) \Leftrightarrow \mathit{false} \qquad [m \neq \texttt{set} \lor n \neq \texttt{set}] \\ &\mathit{LeafMultiProd}(T\ m\texttt{->}n\ Q) \Leftrightarrow \mathit{false} \qquad [m \neq \texttt{set} \lor n \neq \texttt{set}] \\ &\mathit{LeafMultiProd}(T\ m\texttt{->}n\ U) \Leftrightarrow \mathit{false} \end{aligned}$$

where $P, Q : \mathit{FixedArityType}$ and $T, U : \mathit{RelType} \setminus \mathit{FixedArityType}$ and $m, n : \mathit{Multi}$. ◇

Not all type expressions can be written as Class A or B. Those that cannot are often over-constrained. For example, a value in (`X one->one Y`) `one->set Z` contains for each element in the carrier set of `Z` a bijection between the carrier sets of `X` and `Y` which is a singleton. This is only possible if `X` and `Y`'s carrier sets are singletons (or `Z`'s is empty).

Nearly all type expressions that occur in practice are Class A or Class B types, and our theory deals mainly with these types. Even so, we can deal with type expressions which are not of Class A or Class B. By substituting the keyword `set`, for the keywords `one`, `lone`, and `some`, as appropriate in such type expressions they can be made into ones which can be written as Class A or B. Although these substitutions change the meaning of the type expressions, they only increase the possible set of values in the meaning of the type expression (see Definition 2.14), so the calculated contribution will suffice, even though sometimes producing greater thresholds than necessary.

We now define a function which we shall use shortly in the calculation of the contribution of a type which includes `one` or `lone` keywords. The domain of the function is the fixed arity types. Values associated with the meaning of these types contain sequences of atoms and in each sequence a number of atoms belong to the carrier set of the data independent type variable. The function returns a bound on this number.

Definition 5.40 Columns

Let $X : \mathit{TypeVar}$. We define $\mathit{Columns}_X : \mathit{FixedArityType} \to \mathbb{N}$ as follows:

$$\mathit{Columns}_X(Y) = (\textbf{if } X = Y \textbf{ then } 1 \textbf{ else } 0)$$
$$\mathit{Columns}_X(P \texttt{ + } Q) = \mathit{max}\{\mathit{Columns}_X(P), \mathit{Columns}_X(Q)\}$$
$$\mathit{Columns}_X(P \texttt{ -> } Q) = \mathit{Columns}_X(P) + \mathit{Columns}_X(Q)$$

$\diamond$

Lemma 5.41

Let Υ be a set of type variables. Let $X : \Upsilon$. Let $\delta : \mathit{SetMap}(\Upsilon)$. Let $P : \mathit{FixedArityType}$ and suppose $\mathit{Used}(P) \subseteq \Upsilon$. Then:

$$\forall\, vs : [P]_\delta \bullet \#\{i : \mathrm{dom}\, vs \mid vs(i) \in \delta(X)\} \leq \mathit{Columns}_X(P)$$

$\diamond$

Proof We proceed by induction over the structure of P. Let $vs : [P]_\delta$.

Case Y (where $Y : \mathit{Typevar}$).

Sub-case $Y \neq X$. Note $\delta(X) \cap \delta(Y) = \emptyset$.

$vs \in [Y]_\delta$
$\Rightarrow$ [2.10]
$vs \in \{y : \delta(Y) \bullet \langle y \rangle\}$
$\Rightarrow$
$\mathrm{dom}\, vs = \{1\} \wedge vs(1) \in \delta(Y)$
$\Rightarrow$
$\{i : \mathrm{dom}\, vs \mid vs(i) \in \delta(X)\} = \emptyset$
$\Rightarrow$
$\#\{i : \mathrm{dom}\, vs \mid vs(i) \in \delta(X)\} = 0 \leq \mathit{Columns}_X(Y)$

Sub-case $Y = X$.

$vs \in [X]_\delta$
$\Rightarrow$ [2.10]
$vs \in \{x : \delta(X) \bullet \langle x \rangle\}$
$\Rightarrow$
$\mathrm{dom}\, vs = \{1\} \wedge vs(1) \in \delta(X)$
$\Rightarrow$
$\{i : \mathrm{dom}\, vs \mid vs(i) \in \delta(X)\} = \{1\}$
$\Rightarrow$
$\#\{i : \mathrm{dom}\, vs \mid vs(i) \in \delta(X)\} = 1 \leq \mathit{Columns}_X(X)$

Case P + Q.

$vs \in [P]_\delta \vee vs \in [Q]_\delta$
$\Rightarrow$ [ind. hyp.]
$\#\{i : \mathrm{dom}\, vs \mid vs(i) \in \delta(X)\} \leq \mathit{Columns}_X(P)$
$\quad \vee \#\{i : \mathrm{dom}\, vs \mid vs(i) \in \delta(X)\} \leq \mathit{Columns}_X(Q)$
$\Rightarrow$
$\#\{i : \mathrm{dom}\, vs \mid vs(i) \in \delta(X)\} \leq \mathit{max}\{\mathit{Columns}_X(P), \mathit{Columns}_X(Q)\}$
$\quad = \mathit{Columns}_X(P \texttt{ + } Q)$

Case P `->` Q.
Let $vs : [P \texttt{ -> } Q]_\delta$. Choose $xs : [P]_\delta$; $ys : [Q]_\delta$ such that $vs = xs \frown ys$.

$\#\{i : \mathrm{dom}\, vs \mid vs(i) \in \delta(X)\}$
$=$
$\#\{i : \mathrm{dom}\, xs \mid xs(i) \in \delta(X)\} + \#\{i : \mathrm{dom}\, ys \mid ys(i) \in \delta(X)\}$
$\leq$ [ind. hyp]
$\mathit{Columns}_X(P) + \mathit{Columns}_X(Q)$
$=$
$\mathit{Columns}_X(P \texttt{ -> } Q)$

□

We are now in a position to define a function which maps type expressions to contributions. We first define the function on the domain *ClassA* and then on *ClassB*.

Definition 5.42 Bound of a Class A or Class B type
Let Υ be a set of type variables. Let $\Theta : \mathit{Scope}(\Upsilon)$. Let $X : \Upsilon$. Let $T : \mathit{ClassA} \cup \mathit{ClassB}$ and suppose $\mathit{Used}(T) \subseteq \Upsilon$. We now define $\mathit{Bound}_{X,\Theta} : \mathit{TypeExp} \nrightarrow \mathit{Contribution}$.

Case $T : \mathit{ClassA}$. Write $T = n\ P$ where $n : \mathit{Multi}$ and $P : \mathit{FixedArityType}$. $\mathit{Bound}_{X,\Theta}(n\ P)$ is defined in the following table according to the value of n:

$n =$ `set` $\vee$ $n =$ `some`	$(0, \mathit{Regions}_{X,\Theta}(P))$
$n =$ `one` $\vee$ $n =$ `lone`	$(\mathit{Columns}_X(P), 1)$

Case $T : \mathit{ClassB}$. Write $T =$ `set` S where $S : \mathit{RelType}$. We define $\mathit{Bound}_{X,\Theta}(\texttt{set}\ S) = \mathit{RelBound}_{X,\Theta}(S)$ where $\mathit{RelBound}_{X,\Theta}$ is defined below in Definition 5.50. ◇

We now prove that a value of a Class A type expression induces a partition on the carrier set of the data independent type variable, which belongs to the meaning of the contribution assigned to the type expression by *Bound*.

Lemma 5.43

Let Υ be a set of type variables. Let $\Theta : \mathit{Scope}(\Upsilon)$. Let $X : \Upsilon$. Let $T : \mathit{ClassA}$ and suppose $\mathit{Used}(T) \subseteq \Upsilon$. Let $\delta : \mathit{SetMap}$ such that $\operatorname{dom}\delta = \Upsilon$ and $\# \circ \delta \leq \Theta$. Then:

$$\forall\, V : [\![T]\!]_\delta \bullet \delta(X)/\approx_{(X,V)} \in [\![\mathit{Bound}_{X,\Theta}(T)]\!]_{\delta(X)}$$

$\diamond$

Proof Write $T = n\ P$ where $n : \mathit{Multi}$ and $P : \mathit{FixedArityType}$.

Case $n = \texttt{some} \lor n = \texttt{set}$.

Let $V : [\![\texttt{set}\ P]\!]_\delta \cup [\![\texttt{some}\ P]\!]_\delta$. Then:

$$\begin{array}{ll}
& V \in [\![P]\!]_\delta \\
\Rightarrow & [5.29] \\
& \#(\delta(X)/\approx_{(X,V)}) \leq \mathit{Regions}_{X,\Theta}(P) \\
\Rightarrow & [5.33] \\
& \delta(X)/\approx_{(X,V)} \in [\![(0, \mathit{Regions}_{X,\Theta}(P))]\!]_{\delta(X)}
\end{array}$$

Case $n = \texttt{one} \lor n = \texttt{lone}$

Let $V : [\![\texttt{one}\ P]\!]_\delta \cup [\![\texttt{lone}\ P]\!]_\delta$. Either $\#(V) = 0$ or $\#(V) = 1$. If the former then the result is trivial since $V = \emptyset$ and so $\delta(X)/\approx_{(X,V)} = \{\delta(X)\}$.

In the latter case choose $vs : [P]_\delta$ such that $V = \{vs\}$. First define $\mathit{map} : \mathbb{N} \pfun \mathit{Atom}$ as follows:

$$\mathit{map} = \{i : \operatorname{dom} vs \mid vs(i) \in \delta(X) \bullet (i, vs(i))\}$$

We argue as follows. Let $x, y : \delta(X)$:

$$\begin{array}{ll}
& x \approx_{(X,V)} y \\
\Leftrightarrow & [5.1,\ 5.25] \\
& (\forall\, xs, ys : \operatorname{seq} \mathit{Atom} \bullet xs \frown \langle x \rangle \frown ys \in V \Leftrightarrow xs \frown \langle y \rangle \frown ys \in V) \\
\Leftrightarrow & \\
& (\forall\, xs, ys : \operatorname{seq} \mathit{Atom} \bullet xs \frown \langle x \rangle \frown ys = vs \Leftrightarrow xs \frown \langle y \rangle \frown ys = vs) \\
\Leftrightarrow & \\
& (\forall\, i : \operatorname{dom} vs \bullet x = vs(i) \Leftrightarrow y = vs(i)) \\
\Leftrightarrow & \\
& x = y \lor (x \notin \operatorname{ran} vs \land y \notin \operatorname{ran} vs)
\end{array}$$

Then:

$$\begin{array}{l}
\delta(X)/\approx_{(X,V)} \\
= \\
\{x : \delta(X) \bullet \{y : \delta(X) \mid x \approx_{(X,V)} y\}\} \\
=
\end{array}$$

$$
\begin{array}{l}
\{x : \delta(X) \cap \operatorname{ran} vs \bullet \{y : \delta(X) \mid x \approx_{(X,V)} y\}\} \\
\quad \cup \{x : \delta(X) \setminus \operatorname{ran} vs \bullet \{y : \delta(X) \mid x \approx_{(X,V)} y\}\} \\
= \\
\{x : \delta(X) \cap \operatorname{ran} vs \bullet \{y : \delta(X) \mid x = y\}\} \\
\quad \cup \{x : \delta(X) \setminus \operatorname{ran} vs \bullet \{y : \delta(X) \mid y \notin \operatorname{ran} vs\}\} \\
= \\
J \cup K
\end{array}
$$

where:

$$
\begin{array}{l}
J = \{x : \delta(X) \cap \operatorname{ran} vs \bullet \{x\}\} \\
K = \{x : \delta(X) \setminus \operatorname{ran} vs \bullet \delta(X) \setminus \operatorname{ran} vs\}
\end{array}
$$

Now $\delta(X)/\approx_{(X,V)} = J \cup K$ is a partition of $\delta(X)$. K is at most a singleton and every element of J is a singleton, and:

$$
\begin{array}{l}
\#J \\
= \\
\#(\delta(X) \cap \operatorname{ran} vs) \\
= \\
\#(\operatorname{ran} map) \\
\leq \\
\#(\operatorname{dom} map) \\
\leq \\
\mathit{Columns}_X(P)
\end{array}
\qquad [5.41]
$$

This shows:

$$\delta(X)/\approx_{(X,V)} \in [\![(\mathit{Columns}_X(P), 1)]\!]_{\delta(X)}$$

□

We will shortly prove a similar result for Class B type expressions, but before doing so we introduce some subsidiary definitions and lemmas.

The following definition is used to bound the number of sequences of atoms in a value which contain a particular element of the carrier set of the data independent type, by consideration of the type of the value.

Definition 5.44 Rows

Let Υ be a set of type variables. Let $\Theta : \mathit{Scope}(\Upsilon)$. Let $X : \Upsilon$. Let $P : \mathit{FixedArityType}$ and suppose $\mathit{Used}(P) \subseteq \Upsilon$. We now define: $\mathit{Rows}_{(X,\Theta)} : \mathit{FixedArityType} \to \mathbb{N}^\infty$ by recursion:

$$
\begin{array}{l}
\mathit{Rows}_{(X,\Theta)}(Y) = \textbf{if } Y = X \textbf{ then } 1 \textbf{ else } 0 \\
\mathit{Rows}_{(X,\Theta)}(P \mathbin{+} Q) = \mathit{Rows}_{(X,\Theta)}(P) + \mathit{Rows}_{(X,\Theta)}(Q)
\end{array}
$$

where $Y : \mathit{TypeVar}$ and $P, Q : \mathit{FixedArityType}$.

The value of $\mathit{Rows}_{(X,\Theta)}(P \texttt{->} Q)$ is given in the following table:

	$\mathit{Rows}_{(X,\Theta)}(Q) = 0$	$\mathit{Rows}_{(X,\Theta)}(Q) \geq 1$
$\mathit{Rows}_{(X,\Theta)}(P) = 0$	0	$\mathit{Size}_\Theta(P) * \mathit{Rows}_{(X,\Theta)}(Q)$
$\mathit{Rows}_{(X,\Theta)}(P) \geq 1$	$\mathit{Size}_\Theta(Q) * \mathit{Rows}_{(X,\Theta)}(P)$	∞

$\diamond$

Lemma 5.45

Let Υ be a set of type variables. Let $\Theta : \mathit{Scope}(\Upsilon)$. Let $X : \Upsilon$. Let $P : \mathit{FixedArityType}$ and suppose $\mathit{Used}(P) \subseteq \Upsilon$. Then:

$$\mathit{Regions}_{X,\Theta}(P) = 2^{\mathit{Rows}_{(X,\Theta)}(P)}$$

$\diamond$

Proof The proof is an easy induction on the structure of P using the definition of $\mathit{Regions}_{X,\Theta}(P)$ given in Definition 5.28. $\square$

Lemma 5.46

Let Υ be a set of type variables. Let $\Theta : \mathit{Scope}(\Upsilon)$. Let $X : \Upsilon$. Let $P : \mathit{FixedArityType}$ and suppose $\mathit{Used}(P) \subseteq \Upsilon$. Let $\delta : \mathit{SetMap}(\Upsilon)$ and suppose $\# \circ \delta \leq \Theta$. Then:

$$\forall\, x : \delta(X) \bullet \#\{vs : [P]_\delta \mid x \in \operatorname{ran} vs\} \leq \mathit{Rows}_{(X,\Theta)}(P)$$

$\diamond$

Proof The proof is an induction over the structure of P. Let $x : \delta(X)$.

Case Y.

Sub-case $Y \neq X$.

$$\begin{array}{ll}
& \#(\{vs : [Y]_\delta \mid x \in \operatorname{ran} vs\}) \\
= & \qquad [2.10] \\
& \#(\{y : \delta(Y) \mid x \in \operatorname{ran}\langle y \rangle\}) \\
= & \\
& \#(\{y : \delta(Y) \mid x = y\}) \\
= & \qquad [2.6] \\
& 0 \\
\leq & \\
& \mathit{Rows}_{(X,\Theta)}(Y)
\end{array}$$

Sub-case $Y = X$.

$$\#(\{vs : [X]_\delta \mid x \in \operatorname{ran} vs\})$$

$=$ [2.10]
$\#(\{y : \delta(X) \mid x \in \mathrm{ran}\langle y\rangle\})$
$=$
$\#(\{y : \delta(X) \mid x = y\})$
$=$
1
$\leq$
$\mathit{Rows}_{(X,\Theta)}(X)$

Case P + Q.

$\#(\{vs : [P \texttt{ + } Q]_\delta \mid x \in \mathrm{ran}\, vs\})$
$=$ [2.10]
$\#(\{vs : [P]_\delta \cup [Q]_\delta \mid x \in \mathrm{ran}\, vs\})$
$\leq$
$\#(\{vs : [P]_\delta \mid x \in \mathrm{ran}\, vs\}) + \#(\{vs : [Q]_\delta \mid x \in \mathrm{ran}\, vs\})$
$\leq$ [ind. hyp.]
$\mathit{Rows}_{(X,\Theta)}(P) + \mathit{Rows}_{(X,\Theta)}(Q)$
$=$
$\mathit{Rows}_{(X,\Theta)}(P \texttt{ + } Q)$

Case P -> Q.
Sub-case $\mathit{Rows}_{(X,\Theta)}(P) = 0$.

$\#(\{vs : [P \texttt{ + } Q]_\delta \mid x \in \mathrm{ran}\, vs\})$
$=$ [2.10]
$\#(\{vs : [P]_\delta;\ ws : [Q]_\delta \mid x \in \mathrm{ran}(vs \frown ws)\})$
$=$
$\#(\{vs : [P]_\delta\}) * \#(\{ws : [Q]_\delta \mid x \in \mathrm{ran}\, ws\})$
$\leq$
$\mathit{Size}_\Theta(P) * \mathit{Rows}_{(X,\Theta)}(Q)$
$=$
$\mathit{Rows}_{(X,\Theta)}(P \texttt{ -> } Q)$

Sub-case $\mathit{Rows}_{(X,\Theta)}(Q) = 0$. By symmetry.
Sub-case $\mathit{Rows}_{(X,\Theta)}(P) > 0 \wedge \mathit{Rows}_{(X,\Theta)}(P) > 0$. Nothing to prove.

□

Lemma 5.47
Let Υ be a set of type variables. Let Θ : $\mathit{Scope}(\Upsilon)$. Let X : Υ. Let

$P : \mathit{FixedArityType}$ and suppose $\mathit{Used}(P) \subseteq \Upsilon$. Then:

$$\mathit{Rows}_{(X,\Theta)}(P) \neq \infty \Rightarrow \mathit{Columns}_X(P) \leq 1$$

$\diamond$

Proof The proof is an easy induction over the structure of P using the definitions of *Rows* in Definition 5.44 and *Columns* in Definition 5.40. $\square$

Lemma 5.48

Let Υ be a set of type variables. Let $X : \Upsilon$. Let $P : \mathit{FixedArityType}$ and suppose $\mathit{Used}(P) \subseteq \Upsilon$. Let $\delta : \mathit{SetMap}(\Upsilon)$.

Let $xs, ys : \mathrm{seq}\, \mathit{Atom}$. Let $x, y : \delta(X)$. Then:

$$xs \frown \langle x \rangle \frown ys \in [P]_\delta \Rightarrow xs \frown \langle y \rangle \frown ys \in [P]_\delta$$

$\diamond$

Proof The proof is an easy induction over the structure of P using Definition 2.10. $\square$

Definition 5.49 Substitution

Let $xs : \mathrm{seq}\, \mathit{Atom}$. Let $y, z : \mathit{Atom}$. We define $xs[y/z] : \mathrm{seq}\, \mathit{Atom}$ recursively:

$$\begin{array}{l} \langle\rangle[y/z] = \langle\rangle \\ (\langle x \rangle \frown xs)[y/z] = \\ \quad \textbf{if } x = z \textbf{ then } \langle y \rangle \frown xs[y/z] \\ \quad \textbf{else } \langle x \rangle \frown xs[y/z] \end{array}$$

$\diamond$

Now that we have made our subsidiary definitions and lemmas we are ready to define a function which maps an element of *RelType* to a contribution. The contribution tells us the number of singleton blocks and blocks a value whose type is an element of *RelType* will induce on the carrier set of the data independent type via its maximum data relation.

Definition 5.50 Bound of a *RelType*

Let Υ be a set of type variables. Let $\Theta : \mathit{Scope}(\Upsilon)$. Let $X : \Upsilon$. Let $S : \mathit{RelType}$ and suppose $\mathit{Used}(S) \subseteq \Upsilon$. We now define: $\mathit{RelBound}_{X,\Theta}(S) : \mathit{RelType} \nrightarrow \mathit{Contribution}$. The domain of this function shall be: $\{S : \mathit{RelType} \mid \mathit{LeafMultiProd}(S)\}$.

Case $S = P$ where $P : \mathit{FixedArityType}$.

$$\mathit{RelBound}_{X,\Theta}(P) = (0, 2^{\mathit{Rows}_{(X,\Theta)}(P)})$$

Case $S = P\ m\text{->}n\ Q$ where $P, Q : \mathit{FixedArityType}$ and $m, n : \mathit{Multi}$.

We define $\mathit{RelBound}_{X,\Theta}(P\ m\text{->}n\ Q)$ in the following table:

	$Rows_{(X,\Theta)}(Q) = 0$	$Rows_{(X,\Theta)}(Q) > 0$
$Rows_{(X,\Theta)}(P) = 0$	$(0, 1)$	see below
$Rows_{(X,\Theta)}(P) > 0$	see below	$(0, \infty)$

To define the case when $Rows_{(X,\Theta)}(P) = 0$ and $Rows_{(X,\Theta)}(Q) > 0$ we use the following table:

$n = \texttt{set} \vee n = \texttt{some}$	$n = \texttt{one} \vee n = \texttt{lone}$
$(0, (Card_\Theta(m\ P))^{Rows_{(X,\Theta)}(Q)})$	$(Columns_X(Q) * Size_\Theta(P), 1)$

For the case when $Rows_{(X,\Theta)}(Q) = 0$ and $Rows_{(X,\Theta)}(P) > 0$ we define $RelBound_{X,\Theta}(P\ n\texttt{->}m\ Q) = RelBound_{X,\Theta}(Q\ m\texttt{->}n\ P)$, and refer to the table above.

Case $S = P$ `set->set` U where $P : FixedArityType$ and $U : RelType$ and $LeafMultiProd(U)$.

We define $RelBound_{X,\Theta}(P$ `set->set` $U)$ in the following table:

	$RelBound_{X,\Theta}(U) = (0, 1)$	$RelBound_{X,\Theta}(U) \neq (0, 1)$
$Rows_{(X,\Theta)}(P) = 0$	$(0, 1)$	$RelBound_{X,\Theta}(U)^{Size_\Theta(P)}$
$Rows_{(X,\Theta)}(P) > 0$	$(0, Card_\Theta(U)^{Rows_{(X,\Theta)}(P)})$	$(0, \infty)$

Case $S = T$ `set->set` Q where $Q : FixedArityType$ and $T : RelType$ and $LeafMultiProd(T)$.

We define $RelBound_{X,\Theta}(T$ `set->set` $Q) = RelBound_{X,\Theta}(Q$ `set->set` $T)$, and we refer to the previous case. ◇

We now prove a number of lemmas corresponding to the different cases in the above definition of *RelBound*. These lemmas show that the contribution assigned to an element of *RelType* is correct.

Lemma 5.51

Let Υ be a set of type variables. Let $X : \Upsilon$. Let $\Theta : Scope(\Upsilon)$. Let $\delta : SetMap(\Upsilon)$. Suppose $\# \circ \delta \leq \Theta$. Let $S = P$ where $P : FixedArityType$, and suppose $Used(S) \subseteq \Upsilon$. Then:

$$\forall\ V : [\![P]\!]_\delta \bullet \delta(X)/{\approx_{(X,V)}} \in [\![(0, 2^{Rows_{(X,\Theta)}(P)}]\!]_{\delta(X)}$$

◇

Proof

$$\begin{array}{ll} & \#(\delta(X)/{\approx_{(X,V)}}) \\ \leq & [5.29] \\ & Regions_{X,\Theta}(P) \\ = & [5.45] \\ & 2^{Rows_{(X,\Theta)}(P)} \end{array}$$

Thus:

$$(\delta(X)/{\approx_{(X,V)}})$$

$$\in$$
$$[\![(0, 2^{Rows_{(X,\Theta)}(P)}]\!]_{\delta(X)}$$

□

Lemma 5.52

Let Υ be a set of type variables. Let $X : \Upsilon$. Let $\Theta : Scope(\Upsilon)$. Let $\delta : SetMap(\Upsilon)$. Suppose $\# \circ \delta \leq \Theta$. Let $S = P\ m\text{->}n\ Q$ where $P, Q : FixedArityType, m : Multi, n : \{\texttt{set}, \texttt{some}\}$. Suppose also $Used(S) \subseteq \Upsilon$ and $Rows_{(X,\Theta)}(P) = 0$. Then:

$$\forall\, V : [\![P\ m\text{->}n\ Q]\!]_\delta \bullet \delta(X)/\approx_{(X,V)} \in [\![(0, Card_\Theta(m\ P)^{Rows_{(X,\Theta)}(Q)})]\!]_{\delta(X)}$$

◇

Proof We may assume $Rows_{(X,\Theta)}(Q) \neq \infty$ and $\delta(X) \neq \emptyset$ otherwise there is nothing to prove. Let $V : [\![S]\!]_\delta$. Choose $x_0 : \delta(X)$. Let $W = \{ws : [Q]_\delta \mid x_0 \in \operatorname{ran} ws\}$.

$$Rows_{(X,\Theta)}(Q) \neq \infty$$
$$\Rightarrow \qquad [5.47,\ 5.41]$$
$$(\forall\, ws : [Q]_\delta \bullet \#\{i : \operatorname{dom} ws \mid vs(i) \in \delta(X)\} \leq Columns_X(Q) \leq 1)$$
$$\Rightarrow$$
$$(\forall\, ws : [Q]_\delta \bullet \#\{pair : ws \mid second(pair) \in \delta(X)\} \leq 1) \qquad [A]$$
$$\Rightarrow$$
$$(\forall\, x, y : \delta(X);\ ws : [Q]_\delta \bullet ws[x_0/x][y/x_0] = ws[y/x])$$

$$Rows_{(X,\Theta)}(P) = 0$$
$$\Rightarrow \qquad [5.46]$$
$$(\forall\, x : \delta(X) \bullet \#\{vs : [P]_\delta \mid x \in \operatorname{ran} vs\} = 0)$$
$$\Rightarrow$$
$$(\forall\, vs : [P]_\delta \bullet \#\{pair : vs \mid second(pair) \in \delta(X)\} = 0) \qquad [B]$$

We now define a function $\chi : \delta(X) \to (W \to [\![m\ P]\!]_\delta)$ as follows:

$$\chi(x)(ws) = \{vs : [P]_\delta \mid vs \frown ws[x/x_0] \in V\}$$

With reference to Lemmas 5.47 and 5.48:

$$\forall\, ws : W;\ x : \delta(X) \bullet ws[x/x_0] \in [Q]_\delta$$

And so with reference to Definition 2.18 we confirm that $\chi(x)(ws) \in [\![m\ P]\!]_\delta$.

Let $x, y : \delta(X)$. We claim:

$$\chi(x) = \chi(y) \Rightarrow x \approx_{(X,V)} y$$

Let $xs, ys : \mathrm{seq}\, Atom$ and suppose: $xs \frown \langle x \rangle \frown ys \in V$. Choose $vs : [P]_\delta$; $ws : [Q]_\delta$ such that $xs \frown \langle x \rangle \frown ys = vs \frown ws$. From A and B above it follows:

$$\#\{pair : vs \frown ws \mid second(pair) \in \delta(X)\} \leq 1$$
$$\Rightarrow$$
$$\#\{pair : xs \frown \langle y \rangle \frown ys \mid second(pair) \in \delta(X)\} \leq 1$$

Hence we can establish:

$$\mathrm{ran}\, xs \cap \delta(X) = \emptyset \wedge \mathrm{ran}\, ys \cap \delta(X) = \emptyset$$

Thus:

$$xs \frown \langle x \rangle \frown ys = vs \frown ws = vs \frown ws[x_0/x][x/x_0] \in V$$
$$\Rightarrow$$
$$vs \in \chi(x)(ws[x_0/x])$$
$$\Rightarrow$$
$$vs \in \chi(y)(ws[x_0/x])$$
$$\Rightarrow$$
$$vs \frown ws[x_0/x][y/x_0] = vs \frown ws[y/x] \in V$$
$$\Rightarrow$$
$$xs \frown \langle y \rangle \frown ys \in V$$

We can similarly show that:

$$xs \frown \langle y \rangle \frown ys \in V \Rightarrow$$
$$xs \frown \langle x \rangle \frown ys \in V$$

Hence we have established our claim that:

$$\chi(x) = \chi(y) \Rightarrow x \approx_{(X,V)} y$$

It follows that there exists a surjection from $\mathrm{ran}\, \chi$ on to $\delta(X)/\approx_{(X,V)}$. Thus:

$$\#(\delta(X)/\approx_{(X,V)})$$
$$\leq$$
$$\#(\mathrm{ran}\, \chi)$$
$$=$$
$$\#(W \to [\![m\ P]\!]_\delta)$$
$$=$$
$$(\#[\![m\ P]\!]_\delta)^{\#W}$$
$$\leq \qquad [5.27,\ 5.46]$$

$$Card_\Theta(m\ P)^{Rows_{(X,\Theta)}(Q)}$$

□

Lemma 5.53

Let Υ be a set of type variables. Let $X : \Upsilon$. Let $\Theta : Scope(\Upsilon)$. Let $\delta : SetMap(\Upsilon)$. Suppose $\# \circ \delta \leq \Theta$. Let $S = P\ m\text{->}n\ Q$ where $P, Q : FixedArityType, m : Multi, n : \{\texttt{one}, \texttt{lone}\}$. Suppose also $Used(S) \subseteq \Upsilon$ and $Rows_{(X,\Theta)}(P) = 0$. Then:

$$\forall\, V : [\![P\ m\text{->}n\ Q]\!]_\delta \bullet \delta(X)/\approx_{(X,V)} \in [\![(Columns_X(Q) * Size_\Theta(P), 1)]\!]_{\delta(X)}$$

◇

Proof Let $V : [\![S]\!]_\delta$. Let $W = \{vs : [P]_\delta;\ ws : [Q]_\delta \mid vs \frown ws \in V \bullet ws\}$. Given $ws : W$, define:

$$V_{ws} = \{vs : [Q]_\delta \mid vs \frown ws \in V\}$$

It follows:

$$\# W \leq Size_\Theta(P)$$
$$\forall\, ws : W \bullet V_{ws} \in [\![n\ Q]\!]_\delta$$

In exactly the same way as in Theorem 5.29, it follows:

$$\bigcap\{ws : W \bullet \approx_{(X,V_{ws})}\} \subseteq \approx_{(X,V)}$$

Let $C = Bound_{X,\Theta}(n\ Q)$.
We know from Lemma 5.43 that:

$$\forall\, ws : W \bullet \delta(X)/\approx_{(X,V_{ws})} \in [\![C]\!]_{\delta(X)}$$

Hence by Lemma 5.36

$$\delta(X)/\approx_{(X,V)} = RanFold(\otimes)(\{ws : W \bullet ws \mapsto \delta(X)/\approx_{(X,V_{ws})}\})$$
$$\Rightarrow \qquad [5.36]$$
$$\delta(X)/\approx_{(X,V)} \in [\![RanFold(*)(\{ws : W \bullet ws \mapsto C\})]\!]_{\delta(X)}$$
$$\Rightarrow$$
$$\delta(X)/\approx_{(X,V)} \in [\![C^{Size_\Theta(P)}]\!]_{\delta(X)}$$

$$
\begin{array}{l}
C^{Size_\Theta(P)} \\
= \\
(Bound_{X,\Theta}(n\ Q))^{Size_\Theta(P)} \\
= \\
(Columns_X(Q), 1)^{Size_\Theta(P)} \\
= \\
(Columns_X(Q) * Size_\Theta(P), 1)
\end{array}
$$

□

Lemma 5.54

Let Υ be a set of type variables. Let $X : \Upsilon$. Let $\Theta : Scope(\Upsilon)$. Let $\delta : SetMap(\Upsilon)$. Suppose $\# \circ \delta \leq \Theta$. Let $S = T$ `set->set` Q where $T : RelType$ and $LeafMultiProd(T)$ and $Q : FixedArityType$. Suppose $Used(S) \subseteq \Upsilon$ and $RelBound_{X,\Theta}(T) = (0, 1)$. Then:

$$\forall\ V : [\![T \texttt{ set->set } Q]\!]_\delta \bullet \delta(X)/\approx_{(X,V)} \in [\![(0, Card_\Theta(T)^{Rows_{(X,\Theta)}(Q)})]\!]_{\delta(X)}$$

◇

Proof The proof is very similar to the proof of Lemma 5.52. In that proof replace P by T, replace m and n with `set`, and replace $Rows_{(X,\Theta)}(P) = 0$ with $RelBound_{X,\Theta}(P) = (0, 1)$. The proof then gives the required result. □

Lemma 5.55

Let Υ be a set of type variables. Let $X : \Upsilon$. Let $\Theta : Scope(\Upsilon)$. Let $\delta : SetMap(\Upsilon)$. Suppose $\# \circ \delta \leq \Theta$. Let $S = P$ `set->set` U where $P : FixedArityType, U : ClassB$. Suppose $Used(S) \subseteq \Upsilon$ and $Rows_{(X,\Theta)}(P) = 0$. Suppose also:

$$\forall\ V : [\![U]\!]_\delta \bullet \delta(X)/\approx_{(X,V)} \in [\![RelBound_{X,\Theta}(U)]\!]_{\delta(X)} \qquad \text{[A]}$$

Then:

$$\forall\ V : [\![P \texttt{ set->set } U]\!]_\delta \bullet \delta(X)/\approx_{(X,V)} \in [\![RelBound_{X,\Theta}(U)^{Size_\Theta(P)}]\!]_{\delta(X)}$$

◇

Proof The proof is similar to the proof of Theorem 5.53. In that proof, replace Q by U and m and n by `set`. Also replace $Bound_{X,\Theta}(n\ Q)$ with $RelBound_{X,\Theta}(U)$. Instead of Lemma 5.43, one should use A (above), and the proof is valid. □

Lemma 5.56

Let Υ be a set of type variables. Let $X : \Upsilon$. Let $\Theta : Scope(\Upsilon)$. Let $\delta : SetMap(\Upsilon)$. Suppose $\# \circ \delta \leq \Theta$. Let $S : RelType$ and suppose $LeafMultiProd(S)$. Suppose $Used(S) \subseteq \Upsilon$.

Then:

$$\forall\, V : [\![S]\!]_\delta \bullet \delta(X)/\approx_{(X,V)} \in [\![RelBound_{X,\Theta}(S)]\!]_{\delta(X)}$$

$\diamond$

Proof The proof uses induction on the structure of S. Whereas the base cases are provided for by Lemmas 5.52 through 5.54, the inductive step is provided for by Lemma 5.55 (or their symmetrical counterparts). $\square$

We can use this result to extend 5.43 to cover Class B as well as Class A types now.

Theorem 5.57 Let Υ be a set of type variables. Let $\Theta : \mathit{Scope}(\Upsilon)$. Let $X : \Upsilon$. Let $T : \mathit{ClassA} \cup \mathit{ClassB}$ and suppose $\mathit{Used}(T) \subseteq \Upsilon$. Let $\delta : \mathit{SetMap}$ such that $\operatorname{dom}\delta = \Upsilon$ and $\# \circ \delta \leq \Theta$. Then:

$$\forall\, V : [\![T]\!]_\delta \bullet \delta(X)/\approx_{(X,V)} \in [\![Bound_{X,\Theta}(T)]\!]_{\delta(X)}$$

$\diamond$

We have now shown that the function *Bound* maps type expressions to contributions correctly i.e. if a variable is declared with a type expression, then the partition which that variable induces on the carrier set of the data independent type variable belongs to the meaning of the contribution which *Bound* assigns to the type expression.

Example 5.58

We now give an example application of the above theorem. Let:

$$\begin{aligned}
\Upsilon = &\ \{\texttt{Professor}, \texttt{Student}\} \\
\Theta = &\ \{\texttt{Professor} \mapsto 5, \\
&\ \texttt{Student} \mapsto 2\} \\
\delta = &\ \{\texttt{Professor} \mapsto \{x_1, x_2, x_3, x_4, x_5\}, \\
&\ \texttt{Student} \mapsto \{y_1, y_2\}\}
\end{aligned}$$

The above theorem tells us:

$$\begin{aligned}
\forall\, V : &[\![\texttt{set (Student lone->set Professor)}]\!]_\delta \bullet \\
&\delta(\texttt{Professor})/\approx_{(\texttt{Professor},V)} \in \\
&[\![Bound_{\texttt{Professor},\Theta}(\texttt{set (Student lone->set Professor)})]\!]_{\delta(X)}
\end{aligned}$$

We can calculate $Bound_{\texttt{Professor},\Theta}(\texttt{set (Student lone->set Professor)})$ as

follows:

$$
\begin{array}{l}
\mathit{Bound}_{\texttt{Professor},\Theta}(\texttt{set (Student lone->set Professor)}) \\
= \\
\mathit{RelBound}_{\texttt{Professor},\Theta}(\texttt{Student lone -> set Professor}) \\
= \\
(0, \mathit{Card}_{\Theta}(\texttt{lone Student}))^{\mathit{Rows}_{(X,\Theta)}(\texttt{Professor})} \\
= \\
(0, \Theta(\texttt{Student}) + 1)^1 \\
= \\
(0, 3)
\end{array}
$$

So, for example, if:

$$
\begin{array}{rl}
V = \{ & \langle y_1, x_1 \rangle, \\
& \langle y_1, x_2 \rangle, \\
& \langle y_2, x_3 \rangle, \\
& \langle y_2, x_4 \rangle \}
\end{array}
$$

then $V : [\![\texttt{set (Student lone->set Professor)}]\!]_\delta$ and the theorem correctly predicts that $\delta(\texttt{Professor})/\approx_{(\texttt{Professor}, V)}$ is a partition with at most 3 blocks.

$$\delta(\texttt{Professor})/\approx_{(\texttt{Professor}, V)} = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5\}\}$$

$\diamond$

5.4 Small model theorem

We our now ready to use the previous work of the thesis to give a small model theorem: a theorem which reduces the problem of checking the validity or consistency of a formula at an infinite scope to checking it at a finite scope. This will be the main theorem of the thesis.

We first define the bound of a type map. Recall that a binding for a type map induces a partition on the carrier of the data independent type variable via its maximum data relation. The bound of the type map returns a contribution whose meaning contains the partition. In other words it places a bound on the number of blocks (and number of these which are known to be singletons) in the partition.

Definition 5.59 Bound of a type map

Let $X : \mathit{TypeVar}$. Let (Υ, Γ) be a schema. Let $\Theta : \mathit{Scope}(\Upsilon)$. Suppose $\operatorname{ran}\Gamma \subseteq \mathit{ClassA} \cup \mathit{ClassB}$. We define:

$$\mathit{TMBound}_{X,\Theta}(\Gamma) = \mathit{RanFold}(*)(\mathit{Bound}_{X,\Theta} \circ \Gamma)$$

$\diamond$

Theorem 5.60

Let (Υ, Γ) be a schema. Let $X : \Upsilon$. Suppose $\operatorname{ran}\Gamma \subseteq \mathit{ClassA} \cup \mathit{ClassB}$. Let $\Theta : \mathit{Scope}(\Upsilon)$. Let $\delta : \mathit{SetMap}$. Suppose $\operatorname{dom}\delta = \Upsilon$ and $\# \circ \delta \leq \Theta$. Let $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$. Then:

$$\delta(X)/\approx_{(X,\eta)} \in [\![\mathit{TMBound}_{X,\Theta}(\Gamma)]\!]_{\delta(X)}$$

$\diamond$

Proof

By Definition 2.27:

$$\begin{array}{ll} & (\forall\, var : \operatorname{dom}\Gamma \bullet \eta(var) \in [\![\Gamma(var)]\!]_\delta) \\ \Rightarrow & [5.57] \\ & (\forall\, var : \operatorname{dom}\Gamma \bullet \delta(X)/\approx_{(X,\eta(var))} \in [\![\mathit{Bound}_{X,\Theta}(\Gamma(var))]\!]_{\delta(X)}) \end{array}$$

Now, by Definition 5.7:

$$\begin{array}{ll} & \approx_{(X,\eta)} = \bigcap \{ var : \operatorname{dom}\Gamma \bullet \approx_{(X,\eta(var))} \} \\ \Rightarrow & [5.21] \\ & \delta(X)/\approx_{(X,\eta)} = \mathit{RanFold}(\otimes)(\{ var : \operatorname{dom}\Gamma \bullet var \mapsto \delta(X)/\approx_{(X,\eta(var))} \}) \\ \Rightarrow & [5.36] \\ & \delta(X)/\approx_{(X,\eta)} \in \\ & \quad [\![\mathit{RanFold}(*)(\{ var : \operatorname{dom}\Gamma \bullet var \mapsto \mathit{Bound}_{X,\Theta}(\Gamma(var)) \})]\!]_{\delta(X)} \\ \Rightarrow & \\ & \delta(X)/\approx_{(X,\eta)} \in [\![\mathit{TMBound}_{X,\Theta}(\Gamma)]\!]_{\delta(X)} \end{array}$$

$\square$

Now follows the small model theorem and its proof. The proof works by showing that if there is a binding which is an instance/counterexample of a data independent formula, then one can take the quotient of this binding by the maximum data relation. This yields a quotient binding which is also an instance/counterexample but is within a small scope.

Theorem 5.61 Small model theorem

Let (Υ, Γ) be a schema. Let $X : \Upsilon$. Let F be a compatible formula which is data independent in X. Let $\Theta : \mathit{Scope}(\Upsilon)$ and suppose $\Theta(X) = \infty$. Let:

$$\Theta_1 = \Theta \oplus \{ X \mapsto (\mathit{first}(\mathit{TMBound}_{X,\Theta}(\Gamma)) + \mathit{second}(\mathit{TMBound}_{X,\Theta}(\Gamma))) \}$$

Then:

$$(F, \Upsilon, \Gamma)\ \mathit{ConsistentWithin}\ \Theta \Leftrightarrow (F, \Upsilon, \Gamma)\ \mathit{ConsistentWithin}\ \Theta_1$$

$\diamond$

Proof The reverse implication follows simply from $\Theta_1 \leq \Theta$. To prove the forward implication suppose (F, Υ, Γ) *ConsistentWithin* Θ. Consequently, one may choose $\delta : SetMap$ and $\eta : Binding(\Upsilon, \Gamma, \delta)$ such that:

$$\begin{array}{l} \mathrm{dom}\,\delta = \Upsilon \\ \# \circ \delta \leq \Theta \\ [\![F]\!]^{\eta}_{\delta} \end{array}$$

Now let $\sim = \sim_{(X,\eta)}$. It follows:

$$\begin{array}{l} \sim \in DataRel_X(\delta) \\ Closed_{\sim}(\eta) \end{array}$$

And hence by the definition of semantic data independence:

$$[\![F]\!]^{\eta/\sim}_{\delta/\sim}$$

But also:

$$\begin{array}{l} (\delta/\sim)(X) \in [\![TMBound_{X,\Theta}(\Gamma)]\!]_{\delta(X)} \\ \Rightarrow \\ \#((\delta/\sim)(X)) \leq first(TMBound_{X,\Theta}(\Gamma)) + second(TMBound_{X,\Theta}(\Gamma)) \end{array}$$

Whereas:

$$\begin{array}{l} (\forall\, Y : \Upsilon \setminus \{X\} \bullet (\delta/\sim)(Y) = \{y : \delta(Y) \bullet \{y\}\}) \\ \Rightarrow \\ (\forall\, Y : \Upsilon \setminus \{X\} \bullet \#((\delta/\sim)(Y)) = \#(\delta(Y)) \leq \Theta_1(Y)) \end{array}$$

This establishes:

$$\# \circ (\delta/\sim) \leq \Theta_1$$

And so (F, Υ, Γ) *ConsistentWithin* Θ_1, completing the proof. □

Example 5.62

We now give an example application of the above theorem. We define a schema (Υ, Γ) by:

$$\begin{array}{rl} \Upsilon = \{\mathtt{X}\} & \\ \Gamma = \{\mathtt{a} & \mapsto (\mathtt{set\ X})\} \\ \quad\ \mathtt{b} & \mapsto (\mathtt{set\ X})\} \\ \quad\ \mathtt{x} & \mapsto (\mathtt{one\ X})\} \end{array}$$

The following formula is compatible:

$$(\mathtt{a\ in\ b})\ \mathtt{and}\ (\mathtt{x\ in}\ (\mathtt{a\ -\ b}))$$

and in this example we want to check the consistency of this formula. We define

the scope:

$$\Theta = \{\mathtt{X} \mapsto \infty\}$$

We want to check the formula at this scope, but this is an infinite scope and so we cannot check it automatically. As the formula only uses data independent constructs it is syntactically data independent and so we can apply the small model theorem.

We first calculate:

$$\mathit{Bound}_{\mathtt{X},\Theta}(\mathtt{set\ X}) = (0, 2)$$
$$\mathit{Bound}_{\mathtt{X},\Theta}(\mathtt{one\ X}) = (1, 1)$$

and hence:

$$\mathit{TMBound}_{\mathtt{X},\Theta}(\Gamma) = (0, 2) * (0, 2) * (1, 1) = (1, 4)$$

The theorem tell us that if the formula is consistent, then it is consistent within the following scope:

$$\Theta_1 = \{\mathtt{X} \mapsto (1 + 4)\}$$

which can be checked automatically. ◇

5.5 Necessity of thresholds

The preceding work in this chapter has established thresholds for the scope of data independent type variables, and we have proved that these thresholds are sufficient to determine the consistency/validity of formulas (within the scope of the other type variables). In this section we give an indication that the thresholds we have developed are necessary i.e. these thresholds can not be reduced if the analysis is to be complete (with respect to the data-independent type variable).

To prove that every threshold that can be calculated with the functions specified in this chapter is necessary would be very laborious. We take the view that work showing thresholds are sufficient is more beneficial than work showing they are necessary, and have therefore concentrated on the former. In this section we pick a couple of cases, and show that the thresholds calculated in these cases can be attained. The constructions used in these examples give an indication of how to construct a more general proof.

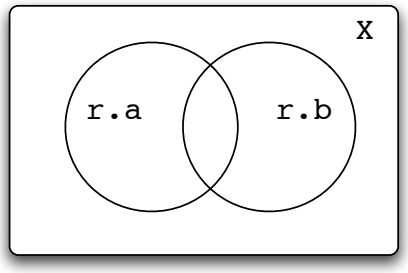
Example 5.63

Consider a formula with a schema (Υ, Γ), where:

$$\begin{aligned}\Upsilon &= \{\texttt{X}, \texttt{Y}\} \\ \Gamma &= \{\texttt{a} \mapsto \ \texttt{one Y}, \\ &\quad\ \texttt{b} \mapsto \ \texttt{one Y}, \\ &\quad\ \texttt{r} \mapsto \ \texttt{set(X -> Y)}\}\end{aligned}$$

If the formula is data independent in `X` and the scope of the type variable `Y` is $\Theta(\texttt{Y}) = 2$, we can calculate a threshold of $\mathit{Regions}_{(\texttt{X},\Theta)}(\texttt{set(X -> Y)}) = 2^{\Theta(Y)} = 4$ for the scope of `X`. We show that this scope is necessary by constructing a formula which is consistent within a scope of 4 on `X` and inconsistent when the scope on `X` is less.

To construct such a formula we observe that each element in the carrier set of `Y` induces a subset of the carrier set of `X` via the relation `r`. Since $\Theta(\texttt{Y}) = 2$ there are two subsets and these give rise to a partition with up to 4 blocks. We show the case of 4 blocks in the following Venn diagram:



The four blocks are represented by the following expressions:

```
(r . a) & (r . b)
(X - (r . a)) & (r . b)
(r . a) & (X - (r . b))
(X - (r . a)) & (X - (r . b))
```

and we construct our formula so that when it is true these expressions have values which are non-empty and mutually disjoint. The formula F, which we

seek, is the conjunction of the following:

```
!((X - X) = ((r . a) & (r . b)))
!((X - X) = ((X - (r . a)) & (r . b)))
!((X - X) = ((r . a) & (X - (r . b))))
!((X - X) = ((X - (r . a)) & (X - (r . b))))
(((r . a) & (r . b)) & ((X - (r . a)) & (r . b))) = (X - X)
(((r . a) & (r . b)) & ((r . a) & (X - (r . b)))) = (X - X)
(((r . a) & (r . b)) & ((X - (r . a)) & (X - (r . b)))) = (X - X)
(((X - (r . a)) & (r . b)) & ((r . a) & (X - (r . b)))) = (X - X)
(((X - (r . a)) & (r . b)) & ((X - (r . a)) & (X - (r . b)))) = (X - X)
(((r . a) & (X - (r . b))) & ((X - (r . a)) & (X - (r . b)))) = (X - X)
```

We now show that the formula is consistent within a scope of 4 for `X`. We define $\delta_0 : \mathit{SetMap}(\Upsilon)$ by:

$$\begin{array}{rl}\delta_0 = \{\mathtt{X} \mapsto \{ & x_1, x_2, x_3, x_4\}, \\ \mathtt{Y} \mapsto \{ & y_1, y_2\}\}\end{array}$$

and $\eta_0 : \mathit{Binding}(\Upsilon, \Gamma, \delta_0)$ by:

$$\begin{array}{rll}\eta_0 = \{\mathtt{a} \mapsto \{ & \langle y_1\rangle\}, \\ \mathtt{b} \mapsto \{ & \langle y_2\rangle\}, \\ \mathtt{r} \mapsto \{ & \langle x_2, y_1\rangle, \\ & \langle x_4, y_1\rangle, \\ & \langle x_3, y_2\rangle, \\ & \langle x_4, y_2\rangle\}\end{array}$$

Then:

$$\begin{array}{rcl}[\![\mathtt{(r\ .\ a)\ \&\ (r\ .\ b)}]\!]^{\eta_0}_{\delta_0} & = & \langle x_4\rangle \\ [\![\mathtt{(X\ -\ (r\ .\ a))\ \&\ (r\ .\ b)}]\!]^{\eta_0}_{\delta_0} & = & \langle x_3\rangle \\ [\![\mathtt{(r\ .\ a)\ \&\ (X\ -\ (r\ .\ b))}]\!]^{\eta_0}_{\delta_0} & = & \langle x_2\rangle \\ [\![\mathtt{(X\ -\ (r\ .\ a))\ \&\ (X\ -\ (r\ .\ b))}]\!]^{\eta_0}_{\delta_0} & = & \langle x_1\rangle\end{array}$$

and it is clear that $[\![F]\!]^{\eta_0}_{\delta_0}$.

To see that the formula is not consistent within a scope of less than 4 on `X`, we suppose $\delta : \mathit{SetMap}(\Upsilon)$ and $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$ such that $[\![F]\!]^{\eta}_{\delta}$. Then we let $A, B, C, D : \mathbb{P}(\delta(\mathtt{X}))$ be defined by:

$$\begin{array}{l}A = \{x : \delta(\mathtt{X}) \mid \langle x\rangle \in [\![\mathtt{(r\ .\ a)\ \&\ (r\ .\ b)}]\!]^{\eta}_{\delta}\} \\ B = \{x : \delta(\mathtt{X}) \mid \langle x\rangle \in [\![\mathtt{(X\ -\ (r\ .\ a))\ \&\ (r\ .\ b)}]\!]^{\eta}_{\delta}\} \\ C = \{x : \delta(\mathtt{X}) \mid \langle x\rangle \in [\![\mathtt{(r\ .\ a)\ \&\ (X\ -\ (r\ .\ b))}]\!]^{\eta}_{\delta}\} \\ D = \{x : \delta(\mathtt{X}) \mid \langle x\rangle \in [\![\mathtt{(X\ -\ (r\ .\ a))\ \&\ (X\ -\ (r\ .\ b))}]\!]^{\eta}_{\delta}\}\end{array}$$

It follows $A \cup B \cup C \cup D = \delta(\mathtt{X})$. The first four conjuncts of the formula F guarantee that the sets A, B, C, and D are each non-empty while the remainder

guarantee they are mutually disjoint. Thus $\{A, B, C, D\}$ is a partition of the carrier set of `X` into 4 blocks. It follows that the carrier set itself must have at least 4 elements.

Note that the threshold in this example does not depend on the variables `a` and `b`. Were they not present in the type map we could still have generated a counterexample formula in a similar way, but by introducing `a` and `b` using existential quantification over `Y`.

$\diamond$

When a data independent formula makes use of a relation over the data independent type variable, the calculation of a threshold yields infinity. In other words we fail to generate a threshold. Calling such formulas *data independent* is perhaps a misnomer, although we did this in the thesis because we separated concerns relating to the semantics and syntax of formulas from concerns about the type of variables used in formulas. The failure to generate thresholds in these cases fits with one's intuition, since elements of a data independent type variable should be compared for equality only, and the declaration of a relation permits other comparisons. The following example includes a relation on the data independent type variable and we show that a threshold can not be generated.

Example 5.64 Consider a formula with a schema (Υ, Γ), where:

$$\begin{array}{ll} \Upsilon = \{\texttt{X}\} & \\ \Gamma = \{\texttt{x} \mapsto & \texttt{one X}, \\ \quad \texttt{r} \mapsto & \texttt{set (X one->one X)}\} \end{array}$$

If the formula is data independent in `X` we can calculate a threshold of ∞ for the scope of `X`. We give a sequence of formulas where formula number n in the sequence is consistent within a scope of $n+1$ for X, but inconsistent within smaller scopes. This shows that it is impossible to give a finite threshold for the carrier set of `X` which is sufficient for all data independent formulas.

Our sequence of formulas, F_n, begins:

$$\begin{array}{ll} F_1 = & \texttt{!(x = (r . x))} \\ F_2 = & \texttt{(!(x = (r . x))) and} \\ & \texttt{((!(x = (r . (r . x)))) and (!((r . x) = (r . (r . x)))))} \end{array}$$

and the general term is defined by the following recursions:

$$e_0 = \texttt{x}$$
$$e_{n+1} = (\texttt{r . } e_n)$$

$$D_{n,0} = (\texttt{! } (e_0 \texttt{ = } e_n))$$
$$D_{n,i+1} = (D_{n,i} \texttt{ and } (\texttt{!}(e_{i+1} \texttt{ = } e_n)))$$

$$E_n = D_{n,n-1}$$

$$F_1 = E_1$$
$$F_{n+1} = (F_n \texttt{ and } E_{n+1})$$

Thus the expression e_n consists of n applications of the function `r` to `x`, and thus refers to a particular element of the carrier set of `X`. The formula F_n is constructed so that when it is true, each of the elements referred to by e_0 through e_n are distinct. This is achieved by the sub-formula E_n asserting that e_n is distinct from each of e_0 up to e_{n-1}, and the overall formula F_n being the conjunction of E_1 through E_n.

It thus follows that F_n can be satisfied, but only when `X` has a carrier set of size $n + 1$ or greater. Hence no finite threshold n can exist since it does not complete the analysis of whether the formula F_n is consistent. $\diamond$

Summary

The aim of this chapter was to derive thresholds for the scope of data-independent type variables. We started by defining the maximum data relation for a value. We gave an explicit definition and showed that the definition gives the coarsest data relation for which the value is closed. We then defined the maximum data relation for a binding as the intersection of the maximum data relations of each value it binds.

We then went on to derive an upper bound on the number of equivalence classes induced by a value's maximum data relation as a function of the type of the value. This allowed us to derive an upper bound on the number of classes induced by a binding which binds values which have types which do not use multiplicity keywords.

For types with multiplicity keywords we introduced the concept of a contribution. The contribution records not just the number of equivalence classes induced by a value's maximum data relation, but the number which are known to be singletons. We saw that by using contribution we could obtain tighter upper bounds on the number of equivalence classes induced by a binding which binds values which have types that use multiplicity keywords. We calculated the contribution of types with multiplicity keywords.

We then used these bounds together with the work in the preceding chapters to generate a small model theorem. This proves that the bounds give rise to a threshold scope at which the analysis of a data-independent formula is complete. Finally we gave some examples to show that the thresholds we generated were attainable and therefore indicated that the thresholds given could not be reduced.

Chapter 6

Enhancements

In this chapter we give a number of enhancements to the small model theorem of the previous chapter. These enhancements increase the range of formulas that can be dealt with and reduce the thresholds which the theorem generates.

6.1 Converting to negation normal form and Skolemizing

When the Alloy Analyzer is asked to check the consistency of a formula the first thing it does is convert the formula into negation normal form and then Skolemize the formula [Jac00]. These techniques provide a means to rewrite a formula without changing its consistency, and at the same time remove quantification constructs from the formula. In order to check the validity of a formula the Analyzer negates the formula and then checks if it is not consistent.

We can also apply these techniques (in respect of the data-independent type variable) and may be able to remove quantification constructs over the data-independent type variable and therefore rewrite a formula which is not syntactically data independent as one which is. These techniques can therefore increase the range of formulas to which the small model theorem can be applied.

For example suppose we want to check the validity of the following formula:

```
all x:X|x = X
```

with compatible schema:

$(\{\texttt{X}\}, \emptyset)$

This formula is not syntactically data independent in `X` because it uses quantification over X. However we can use the above techniques to convert it into one which is data independent in `X`. The above formula is valid exactly when

the following formula is not consistent:

```
! all x:X|x = X
```

We now convert this formula to negation normal form. This means pushing negation inside quantification by using de Morgan's laws. This yields:

```
some x:X|!x = X
```

Skolemization [Ham88] eliminates existentially quantified variables. An existentially quantified variable which is enclosed by no universal quantifiers can be replaced by a scalar (known as a Skolem variable). Skolemization yields the formula:

```
!x = X
```

with compatible schema:

$(\{\texttt{X}\}, \{\texttt{x} \mapsto \texttt{one X}\})$

This formula is syntactically-data-independent and the small model theorem can be applied resulting in a threshold of 2 on `X`, which of course produces a counterexample. This means the original formula is invalid.

To Skolemize existentially quantified variables which are enclosed by universal quantifiers, one instead replaces them by functions (known as Skolem functions). For example, Skolemization of the formula:

```
all y:Y| some x:X|y . r = x
```

yields:

```
all y:Y|y . r = y . x
```

where `r` is now a Skolem function of type `set` (`Y set->one X`).

6.2 Discounting use of type variables as expressions

Recall that type variables can be used as expressions. However it is common to find formulas which do not use this construct. In this case we can reduce our thresholds by 1. We will give an informal explanation of why this is the case, because a detailed formal proof would be long and repetitive.

Consider a binding for a formula and all the values of every variable in this binding. These values use atoms, some of which will be from the carrier set of the data-independent type variable. It is possible that that there are some atoms from the carrier set of the data independent type variable which do not

appear in any value. Such atoms form their own equivalence class with respect to the maximum data relation for a binding (see Definition 5.7), because it is clearly the case that swapping one of these atoms for another one of these atoms in a sequence of atoms does not affect whether a sequence of atoms belongs to any value in the binding: neither the former nor latter sequence is in any value.

Now consider what happens to the of truth the formula if it does not use the data-independent type variable as a subexpression, and if we remove these atoms from the carrier set of the data-independent type variable. It is clear that the binding, which did not use these atoms anyway is still a binding. Furthermore the value of any variable which appears as a subexpression in the formula is not affected, and so it follows that the formula, whose truth depends only on these values is not affected.

This shows that one of the equivalence classes which our previous work on thresholds identified is not required in this circumstance and hence we can subtract one from the result of our previous threshold calculation.

6.3 Rewriting fixed arity types

When forming fixed arity types, the sum operator distributes over the product i.e. $(\texttt{X} + \texttt{Y}) \texttt{->} \texttt{Z}$ has the same meaning as $(\texttt{X} \texttt{->} \texttt{Z}) + (\texttt{Y} \texttt{->} \texttt{Z})$. Furthermore the sum operator is idempotent i.e. for any fixed arity type P: $(P + P)$ and P have the same meaning.

Since the functions we apply to fixed arity types in order to calculate thresholds i.e. *Region*, *Rows* and *Columns* can give a smaller result when applied to P compared to $P + P$ it follows that one can reduce the size of thresholds by first expanding type expressions using the above distributive law and then collecting like terms using the idempotence law.

6.4 Eliminating variables

Some formulas contain conjuncts which give the value of a variable explicitly i.e. when the variable is an operand to the equals operator. One can easily eliminate such variables from these formulas by removing such a conjunct and replacing any occurrences of the variable (which are not bound) with the expression on the other side of the equality construct. This obviously does not change the meaning of the formula but removes the contribution of the variable from the threshold which in some cases can give a substantial reduction.

Summary

In this chapter we have given a number of rules to rewrite formulas before applying the small model theorem, which have a beneficial effect. The application of these rules can be automated. Of course the set of rules could be expanded and this would lead us down the route of a theorem prover. From experience of applying the small model theorem to example problems, the above rules were

found to have a high benefit in reducing thresholds and increasing applicability while being relatively easy to apply, and so form an appropriate set to be used together with the small model theorem.

Chapter 7

Multiple data independent type variables

The theory we have developed so far can deal with formulas which use a single data-independent type variable. In this chapter we look at formulas with more than one data-independent type variable, and show how we can arrive at thresholds which apply simultaneously to these type variables.

7.1 Applying the small model theorem repeatedly

In this section we look at achieving thresholds for multiple data independent type variables by applying our existing small model theorem repeatedly. This approach works in some cases, but for others we have to extend our theory, and we give some examples to illustrate this.

Example 7.1

Suppose we have a schema (Υ, Γ) defined by:

$$\begin{aligned}\Upsilon &= \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}\\ \Gamma &= \{\mathtt{a} \mapsto \ \mathtt{set}\ (\mathtt{X} \texttt{->} \mathtt{Y}),\\ &\quad\ \mathtt{b} \mapsto \ \mathtt{set}\ (\mathtt{Y} \texttt{->} \mathtt{Z}),\\ &\quad\ \mathtt{y} \mapsto \ \mathtt{set}\ \mathtt{Y}\}\end{aligned}$$

and we want to check the validity of some compatible formula. Suppose that the formula is syntactically data independent with respect to `X` and the formula is syntactically data independent with respect to `Z`. Suppose also that there is a finite *a priori* bound of n on `Y`. To make a complete analysis we seek to

establish that the formula is valid within the following scope:

$$\begin{aligned}\Theta = \{\mathtt{X} &\mapsto \infty,\\ \mathtt{Y} &\mapsto n,\\ \mathtt{Z} &\mapsto \infty\}\end{aligned}$$

If we apply the small model theorem with `X` as the data-independent type, we obtain a threshold of 2^n on `X` due to the type of `a` which is `set (X -> Y)`. Thus we can establish that checking the formula within the following scope is sufficient:

$$\begin{aligned}\Theta_1 = \{\mathtt{X} &\mapsto 2^n,\\ \mathtt{Y} &\mapsto n,\\ \mathtt{Z} &\mapsto \infty\}\end{aligned}$$

We can then apply the small model theorem a second time, now with `Z` as the data-independent type variable and we can similarly establish that checking the formula within the following scope is sufficient:

$$\begin{aligned}\Theta_2 = \{\mathtt{X} &\mapsto 2^n,\\ \mathtt{Y} &\mapsto n,\\ \mathtt{Z} &\mapsto 2^n\}\end{aligned}$$

Thus we can establish thresholds simultaneously on `X` and `Z` and the validity of the formula can be checked using a finite scope. ◇

The above example shows how we can use the small model theorem repeatedly to give simultaneous thresholds on multiple data independent type variables. So long as there are no variables which are relations between the data-independent type variables this technique works, since the thresholds for the different data-independent type variables can be calculated independently. However the this technique breaks down as soon as the formula uses a relation between more than one data-independent type variable, as we now illustrate.

Example 7.2
Now suppose we have a schema (Υ, Γ) defined by:

$$\begin{aligned}\Upsilon &= \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}\\ \Gamma &= \{\mathtt{a} \mapsto \mathtt{set}\ (X\ \mathtt{->}\ Y),\\ &\quad\ \mathtt{b} \mapsto \mathtt{set}\ (Y\ \mathtt{->}\ Z),\\ &\quad\ \mathtt{c} \mapsto \mathtt{set}\ (X\ \mathtt{->}\ Z)\}\end{aligned}$$

Suppose we have a compatible formula which is syntactically data independent in `X` and `Z` which we want to check for validity. We have an *a priori* bound of n on `Y` and so we want to check the formula is valid within the following

scope:

$$\Theta = \{\texttt{X} \mapsto \infty, \\ \texttt{Y} \mapsto n, \\ \texttt{Z} \mapsto \infty\}$$

Unfortunately on applying the small model theorem with `X` as the data-independent type variable, we cannot reduce the scope of `X`. This is due to the variable `c` which has type `set (X -> Z)` and has a contribution:

$$(0, 2^{\Theta(Z)}) = (0, 2^{\infty}) = (0, \infty)$$

Applying the small model theorem to `Z` has the same problem.

On choosing a scope for `Z` which is finite we can obtain a finite threshold for the scope on `X`, and vice versa, but it is not possible to obtain simultaneous thresholds on `X` and `Z` within the current theory. ◇

This example shows how obtaining thresholds on multiple data-independent type variables, for formulas which use variables which involve relations between these type variables, is not possible with the small model theorem as it stands. Inspection of the functions used to calculate thresholds reveals that applying the small model theorem in respect of one data-independent type variable when there are variables which use, in addition, another data-independent type variable results in an infinite contribution, since the scope on this latter data independent type variable is infinite.

In summary, the approach of repeatedly applying the small model theorem works, but only when there are no variables whose types involve a relation between multiple data-independent type variables. For these other cases we need to extend our theory.

7.2 Extending the small model theorem

Recall from Chapter 5 that the small model theorem works by taking the quotient of any binding of the formula in question by an equivalence relation with respect to which the binding is closed. The contributions arise because each variable's value must be closed with respect to the equivalence relation: using too coarse an equivalence relation will loose information and evaluation of the formula with the quotient binding may not be the same as evaluating it with the original binding.

However in some cases we can use an equivalence relation which is not closed with respect to a variable's value. Even though some information is lost it turns out that in some contexts the lost information is not relevant to the overall truth of the formula.

Example 7.3

Let (Υ, Γ) be a schema defined by:

$$\begin{aligned}\Upsilon &= \{\mathtt{X}, \mathtt{Y}\} \\ \Gamma &= \{\mathtt{v} \mapsto \ \mathtt{set\ X}, \\ &\quad\ \ \mathtt{r} \mapsto \ \mathtt{set\ (X\ ->\ Y)}\}\end{aligned}$$

We will consider the meaning of the expression `v . r` under the following set map and binding:

$$\begin{aligned}\delta = \{\mathtt{X} \mapsto\ & \{x_1, x_2, x_3, x_4\}, \\ \mathtt{Y} \mapsto\ & \{y_1, y_2, y_3\}\} \\ \eta = \{\mathtt{v} \mapsto \{\ & \langle x_1\rangle, \\ & \langle x_2\rangle\}, \\ \mathtt{r} \mapsto \{\ & \langle x_1, y_1\rangle, \\ & \langle x_1, y_2\rangle, \\ & \langle x_2, y_2\rangle, \\ & \langle x_3, y_2\rangle, \\ & \langle x_4, y_3\rangle\}\}\end{aligned}$$

The meaning of the expression with respect to this set map and binding is:

$$\begin{aligned}[\![\mathtt{v\ .\ r}]\!]^{\eta}_{\delta} = \{\ & \langle y_1\rangle, \\ & \langle y_2\rangle\}\end{aligned}$$

We now take the quotient, but not by the maximum data relation of the whole binding, just the maximum data relation of the value of v.

$$\begin{aligned}\delta/{\sim_{(X,\eta(\mathtt{v}))}} = \{\mathtt{X} \mapsto\ & \{\{x_1, x_2\}, \{x_3, x_4\}\} \\ \mathtt{Y} \mapsto\ & \{\{y_1\}, \{y_2\}, \{y_3\}, \{y_2\}\} \\ \eta/{\sim_{(X,\eta(\mathtt{v}))}} = \{\mathtt{v} \mapsto \{\ & \langle\{x_1, x_2\}\rangle\} \\ \mathtt{r} \mapsto \{\ & \langle\{x_1, x_2\}, \{y_1\}\rangle, \\ & \langle\{x_1, x_2\}, \{y_2\}\rangle, \\ & \langle\{x_3, x_4\}, \{y_2\}\rangle, \\ & \langle\{x_3, x_4\}, \{y_3\}\rangle\}\end{aligned}$$

The meaning of the expression with respect to this quotient set map and quotient binding is:

$$\begin{aligned}[\![\mathtt{v\ .\ r}]\!]^{\eta/{\sim_{(X,\eta(\mathtt{v}))}}}_{\delta/{\sim_{(X,\eta(\mathtt{v}))}}} = \{\ & \langle\{y_1\}\rangle, \\ & \langle\{y_2\}\rangle\}\end{aligned}$$

the product of which is the same as the meaning of the expression with respect to the original set map and original binding.

So even though the value of `r` was not closed for the chosen data relation, and in that sense information was lost, the quotient binding can be used to work out the value of the expression under the original binding. ◇

In order for the above approach to be successful more generally, the sequences

of atoms of the value (of the variable, the contribution of which we seek to ignore) must contain a maximum of one element from the carrier set of the data-independent type variable for which we seek a threshold, and these atoms must be eliminated in the join. (In the above example this condition holds for the value of `r`.) We can see if this condition is satisfied my making the following definition which applies to the type of the value.

Definition 7.4 Head only

Let $X : \mathit{TypeVar}$. We define $\mathit{HeadOnly}_X$ as a predicate over $\mathit{TypeExp}$ as follows:

$$
\begin{array}{l}
\mathit{HeadOnly}_X\, Y \Leftrightarrow \mathit{true} \\
\mathit{HeadOnly}_X(P \texttt{ + } Q) \Leftrightarrow \mathit{HeadOnly}_X(P) \wedge \mathit{HeadOnly}_X(Q) \\
\mathit{HeadOnly}_X(P \texttt{ -> } Q) \Leftrightarrow \mathit{HeadOnly}_X(P) \wedge X \notin \mathit{Used}(Q) \\
\mathit{HeadOnly}_X(m\ T) \Leftrightarrow \mathit{HeadOnly}_X(T) \\
\mathit{HeadOnly}_X(T\ m\texttt{->}n\ U) \Leftrightarrow \mathit{HeadOnly}_X(T) \wedge X \notin \mathit{Used}(U)
\end{array}
$$

◇

Lemma 7.5

Let $T : \mathit{TypeExp}$ and suppose $\mathit{HeadOnly}_X(T)$. Let $\delta : \mathit{SetMap}$ and suppose $\mathit{Used}(T) \subseteq \operatorname{dom}\delta$ and $X \in \operatorname{dom}\delta$. Then:

$$\forall\, V : [\![T]\!]_\delta \bullet \forall\, xs : V \bullet \operatorname{ran}(\mathit{tail}(xs)) \cap \delta(X) = \emptyset$$

◇

Proof The proof uses structural induction over T with the statement of the lemma as the inductive hypothesis. □

So a value, the type of which satisfies $\mathit{HeadOnly}_X$ and is used as the right operand of a join, will have all atoms from the carrier set of the data-independent type variable eliminated in the join. We make a symmetrical definition for relations used as the left operand of a join.

Definition 7.6 Last only

Let $X : \mathit{TypeVar}$. We define $\mathit{LastOnly}_X$ as a predicate over $\mathit{TypeExp}$ as follows:

$$
\begin{array}{l}
\mathit{LastOnly}_X\, Y \Leftrightarrow \mathit{true} \\
\mathit{LastOnly}_X(P \texttt{ + } Q) \Leftrightarrow \mathit{LastOnly}_X(P) \wedge \mathit{LastOnly}_X(Q) \\
\mathit{LastOnly}_X(P \texttt{ -> } Q) \Leftrightarrow X \notin \mathit{Used}(P) \wedge \mathit{LastOnly}_X(Q) \\
\mathit{LastOnly}_X(m\ T) \Leftrightarrow \mathit{LastOnly}_X(T) \\
\mathit{LastOnly}_X(T\ m\texttt{->}n\ U) \Leftrightarrow X \notin \mathit{Used}(T) \wedge \mathit{LastOnly}_X(U)
\end{array}
$$

◇

Lemma 7.7

Let $T : \mathit{TypeExp}$ and suppose $\mathit{LastOnly}_X(T)$. Let $\delta : \mathit{SetMap}$ and suppose $\mathit{Used}(T) \subseteq \operatorname{dom}\delta$ and $X \in \operatorname{dom}\delta$. Then:

$$\forall\, V : [\![T]\!]_\delta \bullet \forall\, xs : V \bullet \operatorname{ran}(\mathit{front}(xs)) \cap \delta(X) = \emptyset$$

$\diamond$

A further consideration is that if we want to ignore the contribution of a variable to the overall threshold, then we need to take the quotient of values of that variable, by a data relation with respect to which it is not closed. We need to avoid the danger that the resulting quotient value may not be of the correct type, as we showed in Remark 3.25, as this quotient value may fail to satisfy some multiplicity constraint. We therefore define a type to be *malleable* if the quotient of its values by *any* data relation are of the correct type.

Definition 7.8 Malleable types

Let $X : \mathit{TypeVar}$ and let $T : \mathit{TypeExp}$. We say that T is malleable (w.r.t. X) and write $\mathit{Malleable}(X, T)$ exactly when:

$$\begin{array}{l}\forall\, \delta : \mathit{SetMap} \mid \mathit{Used}(T) \subseteq \operatorname{dom}\delta \bullet \\ \quad \forall \sim : \mathit{DataRel}_X(\delta) \bullet \\ \quad\quad \forall\, V : [\![T]\!]_\delta \bullet \\ \quad\quad\quad V/\!\sim \in [\![T]\!]_{\delta/\sim}\end{array}$$

$\diamond$

We are only interested in malleable types which also satisfy Definition 7.4 or Definition 7.6 (since we want to eliminate atoms of the data-independent type in a join). The following Lemma establishes that some of the types which satisfy these definitions are also malleable.

Lemma 7.9 Malleable types

Let $X : \mathit{TypeVar}$ and $m : \mathit{Multi}$. Let $P : \mathit{FixedArityType}$. Suppose $X \notin \mathit{Used}(P)$. Then:

$$\begin{array}{l}\mathit{Malleable}(X, m\ X) \\ \mathit{Malleable}(X, \texttt{set}\ (X\ m\texttt{->set}\ P)) \\ \mathit{Malleable}(X, \texttt{set}\ (P\ \texttt{set->}m\ X))\end{array}$$

$\diamond$

Proof Firstly, we prove $\mathit{Malleable}(X, m\ X)$. Let $\delta : \mathit{SetMap}$ and suppose $X \in \operatorname{dom}\delta$. Let $\sim : \mathit{DataRel}_X(\delta)$. Let $V : [\![m\ X]\!]_\delta$. By Definition 2.33 it follows:

$$\begin{array}{ll}V \subseteq [X]_\delta \wedge [\![m]\!](V) & \\ \Rightarrow & [3.8,\text{A.1,A.2}] \\ V/\!\sim \subseteq ([X]_\delta)/\!\sim = [X]_{\delta/\sim} \wedge [\![m]\!](V/\!\sim) & \end{array}$$

$$\Rightarrow \qquad [2.33]$$
$$V/\sim \in [\![m\ X]\!]_{\delta/\sim}$$

Secondly, we prove $Malleable(X, \texttt{set}\ (X\ \texttt{m->set}\ P))$. Note that by symmetry this also proves $Malleable(X, \texttt{set}\ (P\ \texttt{set->}m\ X))$. Let $\delta : SetMap$ and suppose $Used(X\ \texttt{m->set}\ P) \subseteq \operatorname{dom}\delta$. Let $\sim : DataRel_X(\delta)$. Let $V : [\![X\ \texttt{m->set}\ P]\!]_\delta$. By Definition 2.33 we may choose $rel : [X]_\delta \leftrightarrow [P]_\delta$ such that[1], :

$$V = \{pair : rel \bullet first(pair) \frown second(pair)\}$$
$$\forall\, ys : [P]_\delta \bullet rel^\sim(\!|\ \{ys\}\ |\!) \in [\![m\ X]\!]_\delta$$

Let:

$$rel' = \{pair : rel \bullet (first(pair)/\sim, second(pair)/\sim)\}$$
$$V' = V/\sim$$

and it follows by theorem 7.10:

$$rel' \in [X]_{\delta/\sim} \leftrightarrow [P]_{\delta/\sim}$$
$$V' = \{pair' : rel' \bullet first(pair') \frown second(pair')\}$$

We want to establish $V' \in [\![X\ \texttt{m->set}\ P]\!]_{\delta/\sim}$ and so to complete the proof we need to show that:

$$\forall\, ys' : [P]_{\delta/\sim} \bullet (rel')^\sim(\!|\ \{ys'\}\ |\!) \in [\![m\ X]\!]_{\delta/\sim}$$

We first establish:

$$\forall\, ys : [P]_\delta \bullet rel^\sim(\!|\ \{ys\}\ |\!) = \{zs : V \mid tail(zs) = ys \bullet \langle head(zs)\rangle\}$$

by showing equivalence of membership. So let $xs : \operatorname{seq} Atom$ and $ys : [P]_\delta$.

$$xs \in rel^\sim(\!|\ \{ys\}\ |\!)$$
$$\Leftrightarrow$$
$$(\exists\, pair : rel \bullet first(pair) = xs \wedge second(pair) = ys)$$
$$\Leftrightarrow$$
$$(\exists\, zs : V \bullet \langle head(zs)\rangle = xs \wedge tail(zs) = ys)$$
$$\Leftrightarrow$$
$$xs \in \{zs : V \mid tail(zs) = ys \bullet \langle head(zs)\rangle\}$$

Similarly:

$$\forall\, ys' : [P]_{\delta/\sim} \bullet (rel')^\sim(\!|\ \{ys'\}\ |\!) = \{zs' : V' \mid tail(zs') = ys' \bullet \langle head(zs')\rangle\}$$

[1]Note that the symbol $\sim$ is used in two senses in this proof. $\sim$ denotes an element of $DataRel_X(\delta)$, but $rel^\sim$ denotes the inverse of the relation rel.

Now let $ys' : [P]_{\delta/\sim}$. By Theorem A.1, $[P]_{\delta/\sim} = [P]_\delta/\sim$ and we may choose $ys : [P]_\delta$ such that $ys' = ys/\sim$. Then:

$$
\begin{array}{ll}
& rel^\sim(\!|\ \{ys\}\ |\!)/\sim \\
= & \\
& \{zs : V \mid tail(zs) = ys \bullet \langle head(zs)\rangle\}/\sim \\
= & [3.8] \\
& \{zs : V \mid tail(zs) = ys \bullet \langle head(zs)\rangle/\sim\} \\
= & [3.8] \\
& \{zs : V \mid tail(zs) = ys \bullet \langle head(zs/\sim)\rangle\} \\
= & [7.11] \\
& \{zs : V \mid tail(zs)/\sim = ys/\sim \bullet \langle head(zs/\sim)\rangle\} \\
= & \\
& \{zs : V \mid tail(zs/\sim) = ys/\sim \bullet \langle head(zs/\sim)\rangle\} \\
= & \\
& \{zs' : V' \mid tail(zs') = ys' \bullet \langle head(zs')\rangle\} \\
= & \\
& (rel')^\sim(\!|\ \{ys'\}\ |\!)
\end{array}
$$

Recall the first part of the proof where we showed $Malleable(X, m\ X)$ and it now follows:

$$
\begin{array}{l}
rel^\sim(\!|\ \{ys\}\ |\!) \in [\![m\ X]\!]_\delta \\
\Rightarrow \\
(rel^\sim(\!|\ \{ys\}\ |\!))/\sim \in [\![m\ X]\!]_{\delta/\sim} \\
\Rightarrow \\
(rel')^\sim(\!|\ \{ys'\}\ |\!) \in [\![m\ X]\!]_{\delta/\sim}
\end{array}
$$

□

The following theorem and lemma is used in the above proof.

Theorem 7.10
Let T be a type expression. Let δ be a set map such that:

$$Used(T) \subseteq \operatorname{dom} \delta$$

Let $\sim : DataRel(\delta)$. Then:

$$([T]_\delta)/\sim = [T]_{\delta/\sim}$$

◇

Proof See Theorem A.1 □

Lemma 7.11

Let $X : \mathit{TypeVar}$. Let $\delta : \mathit{SetMap}$ and suppose $X \in \operatorname{dom}\delta$. Let $\sim : \mathit{DataRel}_X(\delta)$. Let $vs, ws : \operatorname{seq} \mathit{Atom}$ and suppose $\operatorname{ran} vs \cap \delta(X) = \emptyset$ and $\operatorname{ran} ws \cap \delta(X) = \emptyset$. Then:

$$vs = ws \Leftrightarrow vs/\!\sim = ws/\!\sim$$

$\diamond$

Proof

If $\#(vs) \neq \#(ws)$ then neither side of the above equivalence is true and there is nothing more to prove. Otherwise we can proceed by induction on the size of the sequences vs and ws.

Case $vs = \langle\rangle$ and $ws = \langle\rangle$.

This case follows immediately from Definition 3.8.

Case $vs = \langle y\rangle \frown ys$ and $ws = \langle z\rangle \frown zs$.

$$\begin{array}{ll}
\langle y\rangle \frown ys = \langle z\rangle \frown zs & \\
\Leftrightarrow & \\
\langle y\rangle = \langle z\rangle \wedge ys = zs & \\
\Leftrightarrow & \text{[ind. hyp.]} \\
\langle\{y\}\rangle = \langle\{z\}\rangle \wedge ys/\!\sim = zs/\!\sim & \\
\Leftrightarrow & \text{[3.28, 3.1]} \\
\langle[y]_\sim\rangle = \langle[z]_\sim\rangle \wedge ys/\!\sim = zs/\!\sim & \\
\Leftrightarrow & \\
\langle[y]_\sim\rangle \frown (ys/\!\sim) = \langle[z]_\sim\rangle \frown (zs/\!\sim) & \\
\Leftrightarrow & \text{[3.8]} \\
(\langle y\rangle \frown ys)/\!\sim = (\langle z\rangle \frown zs)/\!\sim &
\end{array}$$

$\square$

We now define the concept of *partially closed* bindings. Because we want to ignore the contribution of some but not all of the variables in a type map from our threshold calculation, some of the variables should be bound to closed values, while others need not be.

Definition 7.12 Bindings which are partially closed

Let $\Gamma : \mathit{TypeMap}$ and let $\delta : \mathit{SetMap}$. Suppose $\forall\, var : \operatorname{dom}\Gamma \bullet \mathit{Used}(\Gamma(var)) \subseteq \operatorname{dom}\delta$. Let $\sim : \mathit{DataRel}(\delta)$. Let $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$.

We want to define what it means for η to be closed on a subset of the variables it binds. Let $\mathit{vars} : \mathbb{P}\operatorname{dom}\Gamma$. We define $\mathit{Closed}_\sim(\eta, \mathit{vars})$, as follows:

$$\mathit{Closed}_\sim(\eta, \mathit{vars}) \Leftrightarrow (\forall\, var : \mathit{vars} \bullet \mathit{Closed}_\sim(\eta(var)))$$

$\diamond$

We are now in a position to give an improved definition of semantic data independence.

Definition 7.13 Semantic Data Independence II

Let (Υ, Γ) be a schema. Let $X : \Upsilon$. Let $e : \mathit{Expr}$ (respectively $F : \mathit{Formula}$). Let $\mathit{vars} : \mathbb{P}(\mathrm{dom}\,\Gamma)$.

We define what is means for (e, Γ) to be *data independent in X with critical variables vars* and write $\Gamma \vdash e;\ X(\mathit{vars})$ below, when $\Gamma \vdash e$. If it is not the case that $\Gamma \vdash e$ then the statement (e, Γ) is data independent in X with critical variables *vars* is nonsense and we leave it undefined.

$\Gamma \vdash e;\ X(\mathit{vars})$ holds exactly when: given any $\delta : \mathit{SetMap}(\Upsilon)$, any $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$, and any $\sim : \mathit{DataRel}_X(\delta)$ such that $\mathit{Closed}_\sim(\eta, \mathit{vars})$ and $\eta/\!\sim \in \mathit{Binding}^\sim(\Upsilon, \Gamma, \delta/\!\sim)$:

$$[\![e]\!]^\eta_\delta = [\![e]\!]^{\eta/\sim}_{\delta/\sim}.\sim$$

(respectively

$$[\![F]\!]^\eta_\delta \Leftrightarrow [\![F]\!]^{\eta/\sim}_{\delta/\sim}$$

). ◇

We note that this definition is very similar to Definition 3.29, our first definition of semantic data independence. The main difference is that the data relation here may be chosen more liberally as it is required only that the binding be partially closed with respect to it. We also have the extra condition $\eta/\!\sim \in \mathit{Binding}^\sim(\Upsilon, \Gamma, \delta/\!\sim)$: to ensure that the $\eta/\!\sim$ is indeed a valid quotient binding.

Rather like our previous definition of syntactic data independence given in Chapter 4, we now give an inference system which mirrors the type inference system of the language, but additionally allows us to infer that a formula is data independent and which of its variables are critical.

Definition 7.14 Inference system for data independence with critical variables.

$$\frac{}{\Gamma \vdash \mathit{var};\ X(\{\mathit{var}\})}$$

$$\frac{}{\Gamma \vdash Y;\ X(\emptyset)}$$

$$\frac{\Gamma \vdash e;\ X(\mathit{vars})}{\Gamma \vdash \tilde{}\ e;\ X(\mathit{vars})}$$

$$\frac{\Gamma \vdash e;\ X(\mathit{vars})}{\Gamma \vdash \hat{}\ e;\ X(\mathit{vars})}$$

$$\frac{\Gamma \vdash e;\ X(vars1) \wedge \Gamma \vdash f;\ X(vars2)}{\Gamma \vdash e\ \texttt{+}\ f;\ X(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash e;\ X(vars1) \wedge \Gamma \vdash f;\ X(vars2)}{\Gamma \vdash e\ \texttt{\&}\ f;\ X(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash e;\ X(vars1) \wedge \Gamma \vdash f;\ X(vars2)}{\Gamma \vdash e\ \texttt{-}\ f;\ X(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash e;\ X(vars1) \wedge \Gamma \vdash f;\ X(vars2)}{\Gamma \vdash e\ \texttt{.}\ f;\ X(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash e;\ X(vars1) \wedge \Gamma \vdash var;\ X(\{var\})}{\Gamma \vdash e\ \texttt{.}\ var;\ X(vars1)}\ [\ HeadOnly_X(\Gamma(var)) \wedge Malleable(X, \Gamma(var))\ ]$$

$$\frac{\Gamma \vdash var;\ X(\{var\}) \wedge \Gamma \vdash f;\ X(vars2)}{\Gamma \vdash var\ \texttt{.}\ f;\ X(vars2)}\ [\ LastOnly_X(\Gamma(var)) \wedge Malleable(X, \Gamma(var))\ ]$$

$$\frac{\Gamma \vdash e;\ X(vars1) \wedge \Gamma \vdash f;\ X(vars2)}{\Gamma \vdash e\ \texttt{->}\ f;\ X(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash e;\ X(vars1) \wedge \Gamma \vdash f;\ X(vars2)}{\Gamma \vdash e\ \texttt{in}\ f;\ X(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash e;\ X(vars1) \wedge \Gamma \vdash f;\ X(vars2)}{\Gamma \vdash e\ \texttt{=}\ f;\ X(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash F;\ X(vars1) \wedge \Gamma \vdash G;\ X(vars2)}{\Gamma \vdash F\ \texttt{and}\ G;\ X(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash F;\ X(vars1) \wedge \Gamma \vdash G;\ X(vars2)}{\Gamma \vdash F\ \texttt{or}\ G;\ X(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash F;\ X(vars1) \wedge \Gamma \vdash G;\ X(vars2)}{\Gamma \vdash F\ \texttt{=>}\ G;\ X(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash F;\ X(vars)}{\Gamma \vdash \texttt{!}F;\ X(vars)}$$

$$\frac{\Gamma \oplus \{var \mapsto (\texttt{one}\ Y)\} \vdash F;\ X(vars)}{\Gamma \vdash \texttt{all}\ var\texttt{:}Y\ \texttt{|}\ F;\ X(vars)}\ [\ X \neq Y\ ]$$

$$\frac{\Gamma \oplus \{var \mapsto (\texttt{one}\ Y)\} \vdash F;\ X(vars)}{\Gamma \vdash \texttt{some}\ var\texttt{:}Y\ \texttt{|}\ F;\ X(vars)}\ [\ X \neq Y\ ]$$

Note that in the above: $\Gamma : \mathit{TypeMap}$; $\mathit{var} : \mathit{Var}$; $\mathit{vars}, \mathit{vars}1, \mathit{vars}2 : \mathbb{P}\, \mathit{Var}$; $e, f : \mathit{Expr}$; $F, G : \mathit{Formula}$; $X, Y : \mathit{TypeVar}$. $\diamond$

In practice, the selection of which the above inference rules should be applied is determined by the syntax of the formula for which the critical variables are to be determined. This is because there is only one inference rule for each language constructor, with the exception of the join constructor. For the join constructor the rules with side conditions should be applied in preference to those without as these result in less critical variables and consequently give rise to smaller thresholds. Both rules with side conditions could apply when two variables are joined and in such cases one should explore which rule will give the lower thresholds by trial and error.

We now prove that the set of variables of a formula, which are deemed to be critical by the above inference system, are indeed the critical ones.

Theorem 7.15 The above inference system is sound $\diamond$

Proof The proofs that the inference rules above, where the critical variables in the consequent contain the critical variables in the antecedents are essentially the same as the proofs found in Chapter 4, concerning data-independent constructors. For example the proof that:

$$\frac{\Gamma \vdash e;\ X(\mathit{vars}1) \wedge \Gamma \vdash f;\ X(\mathit{vars}2)}{\Gamma \vdash e\ \texttt{-}\ f;\ X(\mathit{vars}1 \cup \mathit{vars}2)}$$

proceeds as follows.

Let (Υ, Γ) be a schema. Let $e, f : \mathit{Expr}$. Let $\mathit{vars}1, \mathit{vars}2 : \mathbb{P}\, \mathit{Var}$. Let $X : \Upsilon$. Suppose $\Gamma \vdash e;\ X(\mathit{vars}1)$ and $\Gamma \vdash f;\ X(\mathit{vars}2)$. Suppose also $\Gamma \vdash e\ \texttt{-}\ f$. Let $\delta : \mathit{SetMap}(\Upsilon)$. Let $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$. Let $\sim : \mathit{DataRel}_X(\delta)$. Suppose $\mathit{Closed}_\sim(\eta, \mathit{vars}1 \cup \mathit{vars}2)$ and $\eta/\sim \in \mathit{Binding}^\sim(\Upsilon, \Gamma, \delta/\sim)$. It follows: $\mathit{Closed}_\sim(\eta, \mathit{vars}1)$ and $\Gamma \vdash e$ and hence:

$$[\![e]\!]^\eta_\delta = ([\![e]\!]^{\eta/\sim}_{\delta/\sim}).\sim$$

since $\Gamma \vdash e;\ X(\mathit{vars}1)$. Similarly:

$$[\![f]\!]^\eta_\delta = ([\![f]\!]^{\eta/\sim}_{\delta/\sim}).\sim$$

Hence by Lemma 4.9:

$$[\![e\ \texttt{-}\ f]\!]^\eta_\delta = ([\![e\ \texttt{-}\ f]\!]^{\eta/\sim}_{\delta/\sim}).\sim$$

and thus $\Gamma \vdash e\ \texttt{-}\ f;\ X(\mathit{vars}1 \cup \mathit{vars}2)$.

For two of the above inference rules though, the critical variables of the antecedents are not contained in those of the consequent:

$$\frac{\Gamma \vdash e;\ X(\mathit{vars}1) \wedge \Gamma \vdash \mathit{var};\ X(\{\mathit{var}\})}{\Gamma \vdash e\ .\ \mathit{var};\ X(\mathit{vars}1)}\quad [\ \mathit{HeadOnly}_X(\Gamma(\mathit{var})) \wedge \mathit{Malleable}(X, \Gamma(\mathit{var}))\ ]$$

$$\frac{\Gamma \vdash var;\ X(\{var\}) \wedge \Gamma \vdash f;\ X(vars2)}{\Gamma \vdash var\ .\ f;\ X(vars2)}\ [\ LastOnly_X(\Gamma(var)) \wedge Malleable(X, \Gamma(var))\ ]$$

We will show the first one is valid and the second will follow by symmetry. This will complete the proof of this theorem.

Let (Υ, Γ) be a schema. Let $e : Expr$. Let $var : Var$. Let $vars1 : \mathbb{P}\ Var$. Let $X : \Upsilon$. We assume the antecedent and side conditions of the inference rule:

$$\Gamma \vdash e;\ X(vars1) \wedge \Gamma \vdash var;\ X(\{var\})$$
$$HeadOnly_X(\Gamma(var))$$
$$Malleable(X, \Gamma(var))$$

We may also assume that $\Gamma \vdash e\ .\ var$, otherwise the consequent of the inference rule is undefined. Then, the type system (Definition 2.31) lets us choose $i, j : \mathbb{N}$ such that:

$$\Gamma \vdash e : i$$
$$\Gamma \vdash var : j$$

Next, let $\delta : SetMap(\Upsilon)$, $\eta : Binding(\Upsilon, \Gamma, \delta)$, $\sim : DataRel_X(\delta)$ such that $Closed_\sim(\eta, vars1)$ and $\eta/\sim \in Binding^\sim(\Upsilon, \Gamma, \delta/\sim)$. It follows:

$$[\![e]\!]^\eta_\delta = [\![e]\!]^{\eta/\sim}_{\delta/\sim}.\sim$$
$$\Rightarrow \qquad [3.14]$$
$$[\![e]\!]^\eta_\delta/\sim = [\![e]\!]^{\eta/\sim}_{\delta/\sim}$$

We need to show:

$$[\![e\ .\ var]\!]^\eta_\delta = ([\![e\ .\ var]\!]^{\eta/\sim}_{\delta/\sim}).\sim$$

which we do by showing inclusion in both directions. Let $zs : \mathrm{seq}\, Atom$ and suppose $\#(zs) = i + j - 2$. Choose $xs, ys : \mathrm{seq}\, Atom$ such that $\#(xs) = i - 1$, $\#(ys) = j - 1$, and $zs = xs \frown ys$.

$$zs \in [\![e\ .\ var]\!]^\eta_\delta$$
$$\Rightarrow$$
$$xs \frown ys \in [\![e\ .\ var]\!]^\eta_\delta$$
$$\Rightarrow \qquad [2.33]$$
$$(\exists\, x : Atom \bullet xs \frown \langle x \rangle \in [\![e]\!]^\eta_\delta \wedge \langle x \rangle \frown ys \in \eta(var))$$
$$\Rightarrow \qquad [3.8,\ 3.20]$$
$$(\exists\, x : Atom \bullet$$
$$\quad xs/\sim \frown \langle [x]_\sim \rangle \in [\![e]\!]^\eta_\delta/\sim = [\![e]\!]^{\eta/\sim}_{\delta/\sim} \wedge$$
$$\quad \langle [x]_\sim \rangle \frown (ys/\sim) \in (\eta/\sim)(var))$$

$\Rightarrow$ [2.33]

$(xs/\sim) \frown (ys/\sim) \in [\![e \ . \ var]\!]^{\eta/\sim}_{\delta/\sim}$

$\Rightarrow$ [3.8]

$(xs \frown ys)/\sim \in [\![e \ . \ var]\!]^{\eta/\sim}_{\delta/\sim}$

$\Rightarrow$ [3.9]

$((xs \frown ys)/\sim).\sim \subseteq ([\![e \ . \ var]\!]^{\eta/\sim}_{\delta/\sim}).\sim$

$\Rightarrow$ [3.11]

$zs = xs \frown ys \in ([\![e \ . \ var]\!]^{\eta/\sim}_{\delta/\sim}).\sim$

This shows the forward inclusion.

Now we show the reverse inclusion. Suppose $zs \in ([\![e \ . \ var]\!]^{\eta/\sim}_{\delta/\sim}).\sim$. Then:

$xs \frown ys \in ([\![e \ . \ var]\!]^{\eta/\sim}_{\delta/\sim}).\sim$

$\Rightarrow$ [3.8, 3.14]

$(xs \frown ys)/\sim \in [\![e \ . \ var]\!]^{\eta/\sim}_{\delta/\sim}$

$\Rightarrow$ [3.8]

$(xs/\sim) \frown (ys/\sim) \in [\![e \ . \ var]\!]^{\eta/\sim}_{\delta/\sim}$

It follows by Definition 2.33 that one may choose $x' : Atom^\sim$ such that:

$(xs/\sim) \frown \langle x' \rangle \in [\![e]\!]^{\eta/\sim}_{\delta/\sim}$ [A]

$\langle x' \rangle \frown (ys/\sim) \in (\eta/\sim)(var)$ [B]

Using A, we argue:

$(xs/\sim) \frown \langle x' \rangle \in [\![e]\!]^{\eta/\sim}_{\delta/\sim}$

$\Rightarrow$ [3.9]

$((xs/\sim) \frown \langle x' \rangle).\sim \subseteq [\![e]\!]^{\eta/\sim}_{\delta/\sim}.\sim$

$\Rightarrow$ [7.16]

$\{ws : (xs/\sim).\sim; \ x : x' \bullet ws \frown \langle x \rangle\} \subseteq [\![e]\!]^{\eta}_{\delta}$

$\Rightarrow$ [3.11]

$\{x : x' \bullet xs \frown \langle x \rangle\} \subseteq [\![e]\!]^{\eta}_{\delta}$

$\Rightarrow$

$(\forall\, x : x' \bullet xs \frown \langle x \rangle \in [\![e]\!]^{\eta}_{\delta})$ [C]

Using B, we argue:

$\langle x' \rangle \frown (ys/\sim) \in (\eta/\sim)(var)$

$$\Rightarrow \qquad [3.20]$$
$$\langle x' \rangle \frown (ys/\sim) \in \eta(var)/\sim$$

Using Definition 3.8 one is able to choose $zs : \eta(var)$ such that $\langle x' \rangle \frown (ys/\sim) = zs/\sim$. It follows that $ys/\sim = tail(zs/\sim) = tail(zs)/\sim$. Since $HeadOnly_X(\Gamma(var))$ we have:

$$\begin{array}{ll}
& \mathrm{ran}(tail(zs)) \cap \delta(X) = \emptyset \\
\Rightarrow & [3.18] \\
& \mathrm{ran}(tail(zs)/\sim) \cap (\delta/\sim)(X) = \emptyset \\
\Rightarrow & \\
& \mathrm{ran}(ys/\sim) \cap (\delta/\sim)(X) = \emptyset \\
\Rightarrow & \\
& \mathrm{ran}(ys) \cap \delta(X) = \emptyset
\end{array}$$

Hence by Lemma 7.11, $ys = tail(zs)$. Furthermore:

$$\begin{array}{ll}
& x' = head(zs/\sim) \\
\Rightarrow & \\
& x' = [head(zs)]_\sim \\
\Rightarrow & \\
& head(zs) \in x' \\
\Rightarrow & \\
& (\exists\, x : x' \bullet \langle x \rangle \frown ys \in \eta(var)) \qquad [D]
\end{array}$$

From C and D it follows:

$$\begin{array}{ll}
& (\exists\, x : x' \bullet \langle x \rangle \frown ys \in \eta(var) \wedge xs \frown \langle x \rangle \in [\![e]\!]^\eta_\delta) \\
\Rightarrow & [2.33] \\
& zs = xs \frown ys \in [\![e \ \ . \ \ var]\!]^\eta_\delta
\end{array}$$

Which proves the reverse inclusion. □

Lemma 7.16
Let $\sim : EqRel$. Let $ys' : \mathrm{seq}\, Atom^\sim$ and $y' : Atom^\sim$. Then:

$$(ys' \frown \langle y' \rangle).\sim = \{ys : ys'.\sim;\ y : y' \bullet ys \frown \langle y \rangle\}$$

◇

Proof We proceed by induction on ys'.
Case $ys' = \langle\rangle$.

$$(ys' \frown \langle y' \rangle).\sim$$

$$
\begin{aligned}
&= \\
&(\langle y' \rangle \frown \langle\rangle).\sim \\
&= && [3.9] \\
&\{y : y';\ ys : \langle\rangle.\sim \bullet \langle y \rangle \frown ys\} \\
&= \\
&\{y : y';\ ys : \{\langle\rangle\} \bullet \langle y \rangle \frown ys\} \\
&= \\
&\{y : y' \bullet \langle y \rangle\} \\
&= \\
&\{ys : \{\langle\rangle\};\ y : y' \bullet ys \frown \langle y \rangle\} \\
&= \\
&\{ys : ys'.\sim;\ y : y' \bullet ys \frown \langle y \rangle\}
\end{aligned}
$$

Case $ys' = \langle x' \rangle \frown xs$.

$$
\begin{aligned}
&(ys' \frown \langle y' \rangle).\sim \\
&= \\
&(\langle x' \rangle \frown xs' \frown \langle y' \rangle).\sim \\
&= && [3.9] \\
&\{x : x';\ ys : (xs' \frown \langle y' \rangle).\sim \bullet \langle x \rangle \frown ys\} \\
&= && [\text{ind. hyp.}] \\
&\{x : x';\ ys : \{xs : xs'.\sim;\ y : y' \bullet xs \frown \langle y \rangle\} \bullet \langle x \rangle \frown ys\} \\
&= \\
&\{x : x';\ xs : xs'.\sim;\ y : y' \bullet \langle x \rangle \frown xs \frown \langle y \rangle\} \\
&= \\
&\{ys : \{x : x';\ xs : xs'.\sim \bullet \langle x \rangle \frown xs\};\ y : y' \bullet ys \frown \langle y \rangle\} \\
&= \\
&\{ys : (\langle x' \rangle \frown xs').\sim;\ y : y' \bullet ys \frown \langle y \rangle\} \\
&= \\
&\{ys : ys'.\sim;\ y : y' \bullet ys \frown \langle y \rangle\}
\end{aligned}
$$

□

We can now give an enhanced small model theorem. With this version, contributions to the threshold are counted only for the types of critical variables. The proof of this theorem is very similar to Theorem 5.61.

Theorem 7.17 Small model theorem II

Let (Υ, Γ) be a schema. Let $X : \Upsilon$ and $vars : \mathbb{P}(\operatorname{dom}\Gamma)$. Let F be a compatible formula. Suppose $\Gamma \vdash F;\ X(vars)$. Suppose also:

$$\forall\, var : \operatorname{dom}\Gamma \setminus vars \bullet Malleable(X, \Gamma(var))$$

Let: $\Theta : \mathit{Scope}(\Upsilon)$ and suppose $\Theta(X) = \infty$. Let:

$$\Theta_1 = \Theta \oplus \{X \mapsto (\mathit{first}(\mathit{TMBound}_{X,\Theta}(\mathit{vars} \lhd \Gamma)) + \\ \mathit{second}(\mathit{TMBound}_{X,\Theta}(\mathit{vars} \lhd \Gamma)))\}$$

Then:

$$(F, \Upsilon, \Gamma)\ \mathit{ConsistentWithin}\ \Theta \Leftrightarrow (F, \Upsilon, \Gamma)\ \mathit{ConsistentWithin}\ \Theta_1$$

$\diamond$

Proof The reverse implication follows simply from $\Theta_1 \leq \Theta$. To prove the forwards implication suppose $(F, \Upsilon, \Gamma)\ \mathit{ConsistentWithin}\ \Theta$. Choose $\delta : \mathit{SetMap}$ and $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$ such that:

$$\begin{array}{l} \mathrm{dom}\,\delta = \Upsilon \\ \# \circ \delta \leq \Theta \\ [\![F]\!]^{\eta}_{\delta} \end{array}$$

Now let $\sim = \sim_{(X, \mathit{vars} \lhd \eta)}$. It follows:

$$\begin{array}{l} \sim \in \mathit{DataRel}_X(\delta) \\ \mathit{Closed}_{\sim}(\eta, \mathit{vars}) \end{array}$$

Let $\mathit{var} : \mathrm{dom}\,\Gamma$. We will now show $\eta(\mathit{var})/\sim \in [\![\Gamma(\mathit{var})]\!]_{\delta/\sim}$.
Case $\mathit{var} \in \mathit{vars}$.

$$\begin{array}{ll} & \mathit{Closed}_{\sim}(\eta(\mathit{var})) \\ \Rightarrow & [3.24] \\ & \eta(\mathit{var})/\sim \in [\![\Gamma(\mathit{var})]\!]_{\delta/\sim} \end{array}$$

Case $\mathit{var} \notin \mathit{vars}$.

$$\begin{array}{ll} & \mathit{Malleable}(X, \Gamma(\mathit{var})) \\ \Rightarrow & [7.8] \\ & \eta(\mathit{var})/\sim \in [\![\Gamma(\mathit{var})]\!]_{\delta/\sim} \end{array}$$

Hence $\eta/\sim \in \mathit{Binding}^{\sim}(\Upsilon, \Gamma, \delta/\sim)$ and we may invoke Definition 7.13 to establish:

$$[\![F]\!]^{\eta/\sim}_{\delta/\sim}$$

We also have:

$$\begin{array}{ll} & (\delta/\sim)(X) \in [\![\mathit{TMBound}_{X,\Theta}(\mathit{vars} \lhd \Gamma)]\!]_{\delta(X)} \\ \Rightarrow & [5.60] \end{array}$$

$$\#((\delta/\sim)(X)) \leq \mathit{first}(\mathit{TMBound}_{X,\Theta}(\mathit{vars} \lhd \Gamma)) + \mathit{second}(\mathit{TMBound}_{X,\Theta}(\mathit{vars} \lhd \Gamma))$$

Whereas:

$$(\forall\, Y : \Upsilon \setminus \{X\} \bullet (\delta/\sim)(Y) = \{y : \delta(Y) \bullet \{y\}\})$$
$$\Rightarrow$$
$$(\forall\, Y : \Upsilon \setminus \{X\} \bullet \#((\delta/\sim)(Y)) = \#(\delta(Y)) \leq \Theta_1(Y))$$

This establishes:

$$\# \circ (\delta/\sim) \leq \Theta_1$$

And so (F, Υ, Γ) *Consistent Within* Θ_1, completing the proof. □

The calculation of thresholds in the above theorem takes account of the contribution of the critical variables only and therefore results in lower thresholds.

If we can infer that a variable is not critical to a formula then every use of that variable consists of a join on the data independent type variable. If the variable is a relation then intuitively we can think of the variable, rather than being just a relation *between* the data-independent type variable and other type variables, as being a relation *from* the data-independent type variable *to* the other type variables. That is, since the usage of the variable is a join on the data-independent type variable, it consumes elements of the data-independent type variable (the source type variable) and produces other ones (the target type variables).

When there are multiple data-independent type variables we can apply Theorem 7.17 repeatedly to the different data-independent type variables, to achieve simultaneous thresholds on multiple types, even if there are variables which are relations between these type variables. The variables which are from one data-independent type variable to another in the sense described above, tell us the order of data-independent type variables to use in repeated applications of Theorem 7.17. i.e. if a variable is from `X` to `Y` then it's contribution can be discounted from the threshold for `X` but not `Y`, so we must apply the theorem in respect of `X` first. This obtains a finite scope for `X` and allows us to proceed in the calculation of a finite scope for `Y`.

The variables of a formula which involve relations between data-independent type variables can be used to induce a directed hypergraph over the data-independent type variables. Each variable induces a hyperedge from its source type variable to its target type variables. Clearly, one may obtain bounds simultaneously on a set of data-independent type variables by repeated application of Theorem 7.17 so long as there is no cycle involving these types in the hypergraph.

7.2.1 Dealing with unmalleable types

Our second definition of semantic data independence (Definition 7.13) requires that every variable which is not critical be malleable. It is often the case in practice that this condition is not satisfied. In this subsection we consider a small model theorem, which avoids this condition and can be more applicable in practice.

We do not prove this small model theorem, but state it as a conjecture. This is because it requires the definition of a set of quotient atoms and a quotient operation which differs from those given in Chapter 3. To prove this small model theorem would require at least as much work as that which has so far appeared in this chapter, plus as much work as in Chapter 3 and Appendix A. Our intention is to explain how to construct a theorem which avoids the malleability issue, and looks plausible given the proofs of the other small model theorems.

We begin with an example of the type of problem which does not satisfy the malleability condition but occurs frequently in practice, and a sketch of how to side-step the malleability issue.

Example 7.18

Let:

$$\begin{aligned}
\Upsilon &= \{\texttt{X}, \texttt{Y}\} \\
\Gamma &= \{\texttt{a} \mapsto \ \texttt{one X}, \\
&\qquad \texttt{b} \mapsto \ \texttt{one X}, \\
&\qquad \texttt{f} \mapsto \ \texttt{set(X set->one Y)}\} \\
F &= ((\texttt{a . f}) \ \texttt{=} \ (\texttt{b . f})) \ \texttt{=>} \ (\texttt{a = b})
\end{aligned}$$

Thus (Υ, Γ) is a schema and F is a compatible formula. The formula is data independent with respect to both `X` and `Y`. `f` is a relation between `X` and `Y` (in fact a total function from `X` to `Y`) and every occurrence of `f` in F is the right argument to a join. However, we cannot say that F is data independent in `X` with critical variables `a` and `b`, because the type of `f` is not malleable with respect to `X`.

Consider a set map and binding which is a counterexample to the formula:

$$\begin{aligned}
\delta = \{\texttt{X} \mapsto \{ \ & x_1, x_2, x_3, x_4\}, \\
\texttt{Y} \mapsto \{ \ & y_1, y_2\}\} \\
\eta = \{\texttt{a} \mapsto \{ \ & \langle x_1 \rangle\}, \\
\texttt{b} \mapsto \{ \ & \langle x_2 \rangle\}, \\
\texttt{f} \mapsto \{ \ & \langle x_1, y_1 \rangle, \\
& \langle x_2, y_1 \rangle, \\
& \langle x_3 . y_1 \rangle, \\
& \langle x_4, y_2 \rangle\}
\end{aligned}$$

To generate a small model theorem we usually argue that one can take the quotient of the set map and quotient of the binding to find a counterexample

within the threshold scope. Treating `a` and `b` as critical variables, we can obtain a maximum data relation $\sim$. The quotient of the set map by this data relation is:

$$\begin{array}{rl}\delta/\sim = \{\mathtt{X} \mapsto \{ & \{x_1\}, \{x_2\}, \{x_3, x_4\}\}, \\ \mathtt{Y} \mapsto \{ & \{y_1\}, \{y_2\}\}\}\end{array}$$

And the quotient of the binding by this data relation is:

$$\begin{array}{rll}\eta/\sim = \{\mathtt{a} \mapsto \{ & \langle\{x_1\}\rangle\}, \\ \mathtt{b} \mapsto \{ & \langle\{x_2\}\rangle\}, \\ \mathtt{f} \mapsto \{ & \langle\{x_1\}, \{y_1\}\rangle, \\ & \langle\{x_2\}, \{y_1\}\rangle, \\ & \langle\{x_3, x_4\}, \{y_1\}\rangle, \\ & \langle\{x_3, x_4\}, \{y_2\}\rangle\}\end{array}$$

but this is not a quotient binding because `f` is given a value of the wrong type: the single quotient atom $\{x_3, x_4\}$ relates to two quotient atoms $\{y_1\}$ and $\{y_2\}$ whereas is should relate to exactly one.

The solution to this issue is to remove this offending equivalence class ($\{x_3, x_4\}$) when taking the quotient. This yields the following quotient set map and quotient binding.

$$\begin{array}{rll}\delta' = \{\mathtt{X} \mapsto \{ & \{x_1\}, \{x_2\}\}, \\ \mathtt{Y} \mapsto \{ & \{y_1\}, \{y_2\}\}\} \\ \eta' = \{\mathtt{a} \mapsto \{ & \langle\{x_1\}\rangle\}, \\ \mathtt{b} \mapsto \{ & \langle\{x_2\}\rangle\}, \\ \mathtt{f} \mapsto \{ & \langle\{x_1\}, \{y_1\}\rangle, \\ & \langle\{x_2\}, \{y_1\}\rangle\}\end{array}$$

$\diamond$

For the above approach to work more generally the following three conditions must be satisfied. Firstly, the equivalence classes generated by critical the variables (`a` and `b`) are singletons, so removing non-singleton classes has no affect on their values. Secondly, although type variables may be used as expressions in general (see Definition 2.30), this construct is not present in the formula (F) and so the non-singleton equivalence class is irrelevant (see also Section 6.2 for further explanation). Thirdly the non-critical variables (`f`) are of such a type that, keeping only the sequences of atoms which do not include elements of the non-singleton class and throwing away the others when forming the quotient, results in a quotient value of the correct type.

We now define a *singleton filter* which we use on our maximum data relation to remove non-singleton equivalence classes.

Definition 7.19 Singleton filter

Given an equivalence relation, $\sim : \mathit{EqRel}$, we define the *singleton filter* of $\sim$

written $\overline{\sim}$ as follows:

$$x \overline{\sim} y \Leftrightarrow x = y \wedge (\forall z : Atom \bullet z = x \vee \neg\, x \sim z)$$

$\diamond$

Note that the singleton filter of an equivalence relation over $Atom$ is not generally an equivalence relation over the whole of $Atom$ but rather an equivalence relation over a proper subset of $Atom$ (in fact it is just the identity relation on this subset).

Just as we used an equivalence relation over the whole of $Atom$ to define various quotient and product operations in Chapter 3 we now wish to do the same only with a singleton filter of an equivalence relation. We start by generalizing the definition of quotient atoms so that they can be defined with respect to equivalence relations on proper subsets of $Atom$.

Definition 7.20 Quotient Atoms

In the context of an equivalence relation on $Atom$, to which a singleton filter has been applied we define the set of quotient atoms as follows:

$$Atom^{\overline{\sim}} = \{x : Atom \mid x \in \operatorname{dom} \overline{\sim} \bullet [x]_{\overline{\sim}}\}$$

$\diamond$

Note that since $Atom^{\overline{\sim}}$ is a set consisting of singleton sets of atoms, it is isomorphic to a set of atoms. In fact this set is $\operatorname{dom} \overline{\sim}$. It would appear simpler to use this subset as a basis for quotient values etc. We choose the slightly more complex form as this allows us to reuse the definitions and notation given earlier in the thesis.

Recall the definition of the quotient of a sequence of atoms given in Definition 3.8. We extend this definition so that it applies in the context of singleton filtered equivalence relations, noting that the previous definition is adequate so long as every atom in the sequence belongs to the domain of the singleton filtered equivalence relation. We next generalize the definition of the quotient of a value.

Definition 7.21 Quotient of a value by a singleton filtered equivalence relation

Given a singleton filtered equivalence relation $\overline{\sim}$, we define the quotient of a value V by:

$$V/\overline{\sim} = \{xs : V \mid \operatorname{ran} xs \subseteq \operatorname{dom} \overline{\sim} \bullet xs/\overline{\sim}\}$$

$\diamond$

Our previous definitions of the product given in Definition 3.9 generalize without modification. Indeed all the definitions given in Chapter 3 relating to equivalence relations on $Atom$ generalize to singleton filtered equivalence relations on $Atom$ by replacing occurrences of $\sim$ with $\overline{\sim}$, with the exception of the quotient of a set map.

Definition 7.22 Quotient of a set map by a filtered data relation

Given any set map, δ, and any data relation, $\sim : \mathit{DataRel}(\delta)$, we define the quotient of δ by $\eqsim$ written $\delta/\eqsim$ by:

$$\delta/\eqsim = \{\mathit{maplet} : \delta \bullet \\ (\mathit{first}(\mathit{maplet}), \{x : \mathit{second}(\mathit{maplet}) \mid x \in \operatorname{dom} \eqsim \bullet [x]_{\eqsim}\})\}$$

$\diamond$

We now define what it means for a type to be *brittle*. The type of non-critical variables needs to be brittle for the conjectured small model theorem of this subsection to be applicable.

Definition 7.23 Brittle types

Let $X : \mathit{TypeVar}$ and let $T : \mathit{TypeExp}$. We say that T is brittle (w.r.t. X) and write $\mathit{Brittle}(X, T)$ exactly when:

$$\forall \delta : \mathit{SetMap} \mid \mathit{Used}(T) \subseteq \operatorname{dom} \delta \bullet \\ \forall \sim : \mathit{DataRel}_X(\delta) \bullet \\ \forall V : [\![T]\!]_\delta \bullet \\ V/\eqsim \in [\![T]\!]_{\delta/\eqsim}$$

$\diamond$

Conjecture 7.24 Brittle types

Let $X : \mathit{TypeVar}$ and $m : \mathit{Multi}$. Let $T : \mathit{RelType}$. Suppose $X \notin \mathit{Used}(T)$. Then:

$$\mathit{Brittle}(X, \texttt{set}\ X) \\ \mathit{Brittle}(X, \texttt{set}\ (X\ \texttt{set->}m\ T)) \\ \mathit{Brittle}(X, \texttt{set}\ (T\ m\texttt{->set}\ X))$$

$\diamond$

We now come to our third definition of semantic data independence.

Definition 7.25 Semantic data independence III

Let (Υ, Γ) be a schema. Let $X : \Upsilon$. Let $e : \mathit{Expr}$ (respectively $F : \mathit{Formula}$). Let $\mathit{vars} : \mathbb{P}(\operatorname{dom} \Gamma)$.

We define what is means for (e, Γ) to be *data independent in X with acute variables vars* and write $\Gamma \vdash e;\ \overline{X}(\mathit{vars})$ below, when $\Gamma \vdash e$. If it is not the case that $\Gamma \vdash e$ then the statement (e, Γ) is data independent in X with acute variables *vars* is nonsense and we leave it undefined.

$\Gamma \vdash e;\ \overline{X}(\mathit{vars})$ holds exactly when: given any $\delta : \mathit{SetMap}(\Upsilon)$, any $\eta : \mathit{Binding}(\Upsilon, \Gamma, \delta)$, any $\sim : \mathit{DataRel}_X(\delta)$ such that $\mathit{Closed}_{\eqsim}(\eta, \mathit{vars})$ and $\eta/\eqsim \in \mathit{Binding}^{\eqsim}(\Upsilon, \Gamma, \delta/\eqsim)$:

$$[\![e]\!]^\eta_\delta = [\![e]\!]^{\eta/\eqsim}_{\delta/\eqsim}.\eqsim$$

(respectively

$$[\![F]\!]^{\eta}_{\delta} \Leftrightarrow [\![F]\!]^{\eta/\overline{\sim}}_{\delta/\overline{\sim}}$$

).

$\diamond$

Note that this definition of semantic data independence is identical to the the previous one (Definition 7.13) except instead of using quotient and products by the data relation $\sim$, we use its singleton filtered version $\overline{\sim}$. We now call the variables whose values must be closed, *acute* instead of *critical*. Just as before we define an inference system which allows us to infer when variables of a formula are acute.

Definition 7.26 Inference system for data independence with acute variables.

$$\frac{}{\Gamma \vdash var;\ \overline{X}(\{var\})}$$

$$\frac{}{\Gamma \vdash Y;\ \overline{X}(\emptyset)}\ [\ Y \neq X\ ]$$

$$\frac{\Gamma \vdash e;\ \overline{X}(vars)}{\Gamma \vdash \texttt{\~{}}\ e;\ \overline{X}(vars)}$$

$$\frac{\Gamma \vdash e;\ \overline{X}(vars)}{\Gamma \vdash \texttt{\^{}}\ e;\ \overline{X}(vars)}$$

$$\frac{\Gamma \vdash e;\ \overline{X}(vars1) \wedge \Gamma \vdash f;\ \overline{X}(vars2)}{\Gamma \vdash e\ \texttt{+}\ f;\ \overline{X}(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash e;\ \overline{X}(vars1) \wedge \Gamma \vdash f;\ \overline{X}(vars2)}{\Gamma \vdash e\ \texttt{\&}\ f;\ \overline{X}(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash e;\ \overline{X}(vars1) \wedge \Gamma \vdash f;\ \overline{X}(vars2)}{\Gamma \vdash e\ \texttt{-}\ f;\ \overline{X}(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash e;\ \overline{X}(vars1) \wedge \Gamma \vdash f;\ \overline{X}(vars2)}{\Gamma \vdash e\ \texttt{.}\ f;\ \overline{X}(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash e;\ \overline{X}(vars1) \wedge \Gamma \vdash var;\ \overline{X}(\{var\})}{\Gamma \vdash e\ \texttt{.}\ var;\ \overline{X}(vars1)}\ [\ \mathit{HeadOnly}_X(\Gamma(var)) \wedge \mathit{Brittle}(X, \Gamma(var))\ ]$$

$$\frac{\Gamma \vdash var;\ \overline{X}(\{var\}) \wedge \Gamma \vdash f;\ \overline{X}(vars2)}{\Gamma \vdash var\ \texttt{.}\ f;\ \overline{X}(vars2)}\ [\ \mathit{LastOnly}_X(\Gamma(var)) \wedge \mathit{Brittle}(X, \Gamma(var))\ ]$$

$$\frac{\Gamma \vdash e;\ \overline{X}(vars1) \wedge \Gamma \vdash f;\ \overline{X}(vars2)}{\Gamma \vdash e\ \texttt{->}\ f;\ \overline{X}(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash e;\ \overline{X}(vars1) \wedge \Gamma \vdash f;\ \overline{X}(vars2)}{\Gamma \vdash e\ \texttt{in}\ f;\ \overline{X}(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash e;\ \overline{X}(vars1) \wedge \Gamma \vdash f;\ \overline{X}(vars2)}{\Gamma \vdash e\ \texttt{=}\ f;\ \overline{X}(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash F;\ \overline{X}(vars1) \wedge \Gamma \vdash G;\ \overline{X}(vars2)}{\Gamma \vdash F\ \texttt{and}\ G;\ \overline{X}(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash F;\ \overline{X}(vars1) \wedge \Gamma \vdash G;\ \overline{X}(vars2)}{\Gamma \vdash F\ \texttt{or}\ G;\ \overline{X}(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash F;\ \overline{X}(vars1) \wedge \Gamma \vdash G;\ \overline{X}(vars2)}{\Gamma \vdash F\ \texttt{=>}\ G;\ \overline{X}(vars1 \cup vars2)}$$

$$\frac{\Gamma \vdash F;\ \overline{X}(vars)}{\Gamma \vdash \texttt{!}F;\ \overline{X}(vars)}$$

$$\frac{\Gamma \oplus \{var \mapsto (\texttt{one}\ Y)\} \vdash F;\ \overline{X}(vars)}{\Gamma \vdash \texttt{all}\ var\texttt{:}\,Y\,\texttt{|}\,F;\ \overline{X}(vars)}\ [\ X \neq Y\ ]$$

$$\frac{\Gamma \oplus \{var \mapsto (\texttt{one}\ Y)\} \vdash F;\ \overline{X}(vars)}{\Gamma \vdash \texttt{some}\ var\texttt{:}\,Y\,\texttt{|}\,F;\ \overline{X}(vars)}\ [\ X \neq Y\ ]$$

Note that in the above: $\Gamma : \mathit{TypeMap}$; $var : \mathit{Var}$; $vars, vars1, vars2 : \mathbb{P}\ \mathit{Var}$; $e, f : \mathit{Expr}$; $F, G : \mathit{Formula}$; $X, Y : \mathit{TypeVar}$. $\diamond$

Note that this inference system which concerns acute variables is almost identical to the inference system concerning critical variables (Definition 7.14). The differences are that the special rules which allow us to form a join with a variable without the variable being deemed to be critical/acute, now require that the type of the variable be brittle rather than malleable. We also have an extra side condition on the second inference rule to ensure that the data-independent type variable is not used as an expression within a formula.

We conjecture that this inference system is sound, and conjecture the following proposition is a small model theorem.

Conjecture 7.27 Small model conjecture III

Let (Υ, Γ) be a schema. Let $X : \Upsilon$ and $vars : \mathbb{P}(\mathrm{dom}\,\Gamma)$. Let F be a compatible formula. Suppose $\Gamma \vdash F;\ \overline{X}(vars)$. Suppose also:

$$\forall\, var : \mathrm{dom}\,\Gamma \setminus vars \bullet \mathit{Brittle}(X, \Gamma(var))$$

Let: $\Theta : \mathit{Scope}(\Upsilon)$ and suppose $\Theta(X) = \infty$. Let $i : \mathbb{N}$ and suppose:

$$\mathit{TMBound}_{X,\Theta}(\mathit{vars} \lhd \Gamma) = (i, 1)$$

Let:

$$\Theta_1 = \Theta \oplus \{X \mapsto i\}$$

Then:

$$(F, \Upsilon, \Gamma)\ \mathit{ConsistentWithin}\ \Theta \Leftrightarrow (F, \Upsilon, \Gamma)\ \mathit{ConsistentWithin}\ \Theta_1$$

$\diamond$

This conjecture is very similar to the second small model theorem (Theorem 7.17), and it seems plausible that it can be proved in a similar way. The extra condition $\mathit{TMBound}_{X,\Theta}(\mathit{vars} \lhd \Gamma) = (i, 1)$ is needed to ensure that the type of acute variables is such that the singleton filter has no effect on their quotients.

Example 7.28 Comparison of the small model theorems

Define a schema (Υ, Γ) by:

$$\begin{aligned}\Upsilon &= \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}\\ \Gamma &= \{\mathtt{a} \mapsto\ \mathtt{set}\ (\mathtt{Y} \mathtt{->} \mathtt{X}),\\ &\quad\ \mathtt{b} \mapsto\ \mathtt{set}\ (\mathtt{X}\ \mathtt{one} \mathtt{->} \mathtt{set}\ \mathtt{Z}),\\ &\quad\ \mathtt{c} \mapsto\ \mathtt{set}\ (\mathtt{Y} \mathtt{->} \mathtt{Z})\}\end{aligned}$$

Define a formula F by

$$F = \mathtt{a\ .\ b = c}$$

Define a scope Θ by:

$$\begin{aligned}\Theta = \{&\mathtt{X} \mapsto\ \infty,\\ &\mathtt{Y} \mapsto\ n,\\ &\mathtt{Z} \mapsto\ \infty\}\end{aligned}$$

Suppose we want to check the formula F either for consistency or validity within this scope. The scope is infinite so this is not possible to do with the Alloy Analyzer. We would like to determine finite thresholds on both `Y` and `Z`, but as we explained at the start of the chapter, it is not possible using the first small model theorem. The variable `b`, being a relation between `X` and `Z`, gives rise to an infinite contribution, whether the small model theorem is first applied with `X` as the data-independent type variable, or `Z` as the data-independent type variable.

However, by applying the second small model theorem (Theorem 7.17), we can obtain finite thresholds on `Y` and `Z` simultaneously. Because `b` appears in the formula F in a join construct, and its type is malleable (Definition 7.8) we can eliminate its contribution using the inference system for data independence

with critical variables (Definition 7.14):

$$\frac{}{\Gamma \vdash \mathtt{a};\ X(\{\mathtt{a}\})}$$

$$\frac{}{\Gamma \vdash \mathtt{b};\ X(\{\mathtt{b}\})}$$

$$\frac{\Gamma \vdash \mathtt{a};\ X(\{\mathtt{a}\}) \wedge \Gamma \vdash \mathtt{b};\ X(\{\mathtt{b}\})}{\Gamma \vdash \mathtt{a}\ \mathtt{.}\ \mathtt{b};\ X(\{\mathtt{a}\})}\ [\ \mathit{HeadOnly}_X(\Gamma(\mathtt{b})) \wedge \mathit{Malleable}(X, \Gamma(\mathtt{b}))\ ]$$

$$\frac{}{\Gamma \vdash \mathtt{c};\ X(\{\mathtt{c}\})}$$

$$\frac{\Gamma \vdash \mathtt{a}\ \mathtt{.}\ \mathtt{b};\ X(\{\mathtt{a}\}) \wedge \Gamma \vdash \mathtt{c};\ X(\{\mathtt{c}\})}{\Gamma \vdash \mathtt{a}\ \ \mathtt{.}\ \mathtt{b} = \mathtt{c};\ X(\{\mathtt{a}, \mathtt{c}\})}$$

The contribution of `a` is $(0, 2^n)$ and `c` is $(0, 1)$. By using the second small model theorem we have avoided the troublesome contribution of `b`. We have established that it is sufficient to check the formula within the following scope:

$$\begin{aligned}\Theta_1 = \{\mathtt{X} &\mapsto\ 2^n,\\ \mathtt{Y} &\mapsto\ n,\\ \mathtt{Z} &\mapsto\ \infty\}\end{aligned}$$

We can now apply the first small model theorem (Theorem 5.61) with `Z` as the data-independent type variable. The contribution of `a` is $(0, 1)$, the contribution of `b` is $(0, \Theta_1(X)) = (0, 2^n)$, and the contribution of `c` is $(0, 2^{\Theta_1(Y)}) = (0, 2^n)$. The overall contribution is thus $(0, 2^{2n})$.

We now have a finite scope which is sufficient to check the formula:

$$\begin{aligned}\Theta_2 = \{\mathtt{X} &\mapsto\ 2^n,\\ \mathtt{Y} &\mapsto\ n,\\ \mathtt{Z} &\mapsto\ 2^{2n}\}\end{aligned}$$

In this example the type of `b` was malleable with respect to `Y`. Had it instead been brittle (Definition 7.23), we could have still obtained a finite scope in which to check the formula, however we would have had to use the conjectured third small model theorem (Conjecture 7.27). ◇

Summary

In this chapter we have looked at formulas with multiple data-independent type variables and investigated how to obtain thresholds which apply to these

type variables simultaneously. When there are no relations between the data-independent type variables we can proceed by applying the small model theorem developed in Chapter 5 repeatedly.

When there are relations between data-independent type variables we can establish simultaneous thresholds provided the relations are used in a restricted way. The restrictions imposed allow us to see one of the data-independent type variables as a source type and the others as target types. We have written down two small model theorems to deal with these relations, and proved the first. The second remains a conjecture since its proof would require modification of the work in Chapter 3.

We note that although the theory in this chapter was developed to cope with multiple data-independent type variables, even when a formula has only one data-independent type variable, the enhanced versions of the small model theorem may be of use. This is because they may allow the contribution of some variables to be discounted and therefore give lower thresholds which avoid state explosion.

Chapter 8

Strong data independence

A system is strongly data independent in a type if the only operations permitted on the type are input, output and copying between variables of the type. For example a buffer is strongly data independent in the buffered type. Strong data independence is a special case of the data independence that we have already looked at in the thesis, which is sometimes known as weak data independence. Compared to strong data independence, weak data independence permits in addition equality testing.

One can check a system which is strongly data independent in a type T for all instantiation of the type, by checking when T has only two elements e.g 0 and 1. If such a system has an incorrect behaviour which is dependent on T, it must output an incorrect value of type T. Given the set of permitted operations, there exists a subset of the inputs to the system which have the correct value, whereas the incorrect value was copied from an input outside this subset. By setting all the inputs in this subset to 0 and all the inputs outside this subset to 1 the incorrect behaviour can be recreated with only two elements for T.

Theorems have been developed to show that in some formalisms checking at a scope of 2 is sufficient to verify strongly-data-independent systems e.g. Lazić's semantic study of data independence [Laz99] gives such a theorem for the Communicating Sequential Processes (CSP) [Hoa85] [Ros98] formalism.

Unfortunately we have not been able to produce a small model theorem which applies to strongly-data-independent systems in Alloy. Instead, this chapter explores why this appears to be a difficult task.

8.1 A smaller language

In this section we define an even smaller language than the kernel language of Chapter 2. This language is sufficient to explain the difficulties of developing a small model theorem for strongly-data-independent systems in Alloy. Use of this smaller language has the benefit of shortening and simplifying this chapter.

Definition 8.1 Syntax

The language consists of equality tests between variables and the following logic operations: negation, conjunction, disjunction and implication.

$$
\begin{array}{rcl}
\mathit{SmallForm} & ::= & \mathit{Var}\ \texttt{=}\ \mathit{Var} \\
& | & \texttt{!}\ \mathit{SmallForm} \\
& | & \mathit{SmallForm}\ \texttt{and}\ \mathit{SmallForm} \\
& | & \mathit{SmallForm}\ \texttt{or}\ \mathit{SmallForm} \\
& | & \mathit{SmallForm}\ \texttt{=>}\ \mathit{SmallForm}
\end{array}
$$

Variables will be denoted by a, a_1, a_2 etc. and formulas by A, A_1, A_2 etc.

◇

Definition 8.2 Bindings
A binding is a total function from variables to atoms.

$$\mathit{SmallBinding} = \mathit{Var} \to \mathit{Atom}$$

We shall denote a binding by $\kappa, \kappa_1, \kappa_2$ etc. ◇

Definition 8.3 Semantics
Given $\kappa : \mathit{SmallBinding}$ we define the semantics as follows:

$$
\begin{array}{l}
[\![a_1\ \texttt{=}\ a_2]\!]_\kappa \Leftrightarrow \kappa(a_1) = \kappa(a_2) \\
[\![\texttt{!}\ A]\!]_\kappa \Leftrightarrow \neg\ [\![A]\!]_\kappa \\
[\![A_1\ \texttt{and}\ A_2]\!]_\kappa \Leftrightarrow [\![A_1]\!]_\kappa \wedge [\![A_2]\!]_\kappa \\
[\![A_1\ \texttt{or}\ A_2]\!]_\kappa \Leftrightarrow [\![A_1]\!]_\kappa \vee [\![A_2]\!]_\kappa \\
[\![A_1\ \texttt{=>}\ A_2]\!]_\kappa \Leftrightarrow [\![A_1]\!]_\kappa \Rightarrow [\![A_2]\!]_\kappa
\end{array}
$$

◇

We note that every formula generated by the syntax of the language is given a semantics. In other words there is no type system in this language. Every variable can be considered to be of the same type.

Definition 8.4 Scopes
A scope for this smaller language is a single natural number, $i : \mathbb{N}$. A binding is within the scope of i if the number of atoms in its range is less than or equals to i.

◇

8.2 Strongly-data-independent formulas

In this section we look at which formulas of the language can be considered as modeling a strongly-data-independent system.

Strongly-data-independent systems satisfy the following monotonicity property. Given two test cases which differ only in the values of the data-independent type, and for which the second test case identifies at least the same values that

are identified by the first, any strongly-data-independent system which passes the first test case also passes the second.

We now formalize this property.

Definition 8.5 Partial order on bindings

We define a partial order, denoted $\leq$ on bindings as follows.

$$\kappa_1 \leq \kappa_2 \Leftrightarrow \\ (\forall\, a_1, a_2 : \mathit{Var} \bullet \kappa_1(a_1) = \kappa_1(a_2) \Rightarrow \kappa_2(a_1) = \kappa_2(a_2))$$

$\diamond$

Definition 8.6 Monotone formulas

A formula A is *monotone* provided:

$$\forall\, \kappa_1, \kappa_2 : \mathit{SmallBinding} \mid \kappa_1 \leq \kappa_2 \bullet [\![A]\!]_{\kappa_1} \Rightarrow [\![A]\!]_{\kappa_2}$$

$\diamond$

Definition 8.7 Monotone constructors

A constructor which is a logical operator is defined to be a *monotone constructor* if it yields a monotone formula whenever its operands are monotone.

Since a primitive formula of the form a_1 `=` a_2 is monotone, we define the `=` operator to be a monotone constructor also. $\diamond$

Lemma 8.8 Monotone constructors

The following constructors are monotone: `and`, `or`. $\diamond$

Proof

Let A_1, A_2 be monotone formulas. Let $\kappa_1, \kappa_2 : \mathit{SmallBinding}$ and suppose $\kappa_1 \leq \kappa_2$.

$$[\![A_1 \texttt{ and } A_2]\!]_{\kappa_1} \\ \Rightarrow \\ [\![A_1]\!]_{\kappa_1} \wedge [\![A_2]\!]_{\kappa_1} \\ \Rightarrow \\ [\![A_2]\!]_{\kappa_2} \wedge [\![A_2]\!]_{\kappa_2} \\ \Rightarrow \\ [\![A_1 \texttt{ and } A_2]\!]_{\kappa_2}$$

Thus `and` is a monotone constructor.

$$
\begin{array}{l}
[\![A_1 \texttt{ or } A_2]\!]_{\kappa_1} \\
\Rightarrow \\
[\![A_1]\!]_{\kappa_1} \vee [\![A_2]\!]_{\kappa_1} \\
\Rightarrow \\
[\![A_2]\!]_{\kappa_2} \vee [\![A_2]\!]_{\kappa_2} \\
\Rightarrow \\
[\![A_1 \texttt{ or } A_2]\!]_{\kappa_2}
\end{array}
$$

Thus `or` is a monotone constructor. □

Lemma 8.9 Non-monotone constructors
The following constructors are not monotone: `!`, `=>`. ◇

Proof Choose three distinct variables, say a_1, a_2, a_3. Define κ_1 by $\kappa_1(a_1) = 0$, $\kappa_1(a_2) = 1$, $\kappa_1(a_3) = 2$, and κ_1 applied to any other variables returns 0. Define κ_2 by $\kappa_2(a_1) = 0$, $\kappa_2(a_2) = 0$, $\kappa_2(a_3) = 2$, and κ_2 applied to any other variables returns 0. It then follows that $\kappa_1 \leq \kappa_2$.

It also follows that $[\![\texttt{!}(a_1 \texttt{ = } a_2)]\!]_{\kappa_1}$ but $\neg\ [\![\texttt{!}(a_1 \texttt{ = } a_2)]\!]_{\kappa_2}$, and hence `!` is not a monotone constructor.

Furthermore:

$$
\begin{array}{l}
[\![(a_1 \texttt{ = } a_2) \texttt{ => } (a_1 \texttt{ = } a_3)]\!]_{\kappa_1} \\
\Leftrightarrow \\
[\![(a_1 \texttt{ = } a_2)]\!]_{\kappa_1} \Rightarrow [\![(a_1 \texttt{ = } a_3)]\!]_{\kappa_1} \\
\Leftrightarrow \\
\mathit{false} \Rightarrow \mathit{false} \\
\Leftrightarrow \\
\mathit{true}
\end{array}
$$

but:

$$
\begin{array}{l}
[\![(a_1 \texttt{ = } a_2) \texttt{ => } (a_1 \texttt{ = } a_3)]\!]_{\kappa_2} \\
\Leftrightarrow \\
[\![(a_1 \texttt{ = } a_2)]\!]_{\kappa_2} \Rightarrow [\![(a_1 \texttt{ = } a_3)]\!]_{\kappa_2} \\
\Leftrightarrow \\
\mathit{false} \Rightarrow \mathit{true} \\
\Leftrightarrow \\
\mathit{false}
\end{array}
$$

and hence `=>` is not a monotone constructor. □

Remark 8.10 Although `=>` is not a monotone constructor and should not feature in the description of a strongly-data-independent system, if we want to check the refinement of a specification *SPEC* by an implementation *IMPL* we will need to check the formula *IMPL* `=>` *SPEC*. For strongly-data-independent systems *IMPL* and *SPEC* will be monotone. Therefore the use of `=>` should be permitted for model finding formulas corresponding to checking strongly-data-independent systems, but only at the top level of the syntax tree. ◇

Although monotonicity is a necessary property for strong data independence the following theorem shows that it is not sufficient.

Theorem 8.11 Not all monotone formulas are verifiable with a scope of 2 ◇

Proof

We give an example formula which is monotone and has a counterexample, but no counterexample within a scope of 2.

Let a_1, a_2, a_3 be distinct members of *Var*. We define the example formula A as follows:

$$A = (a_1 \texttt{ = } a_2) \texttt{ or } (a_2 \texttt{ = } a_3) \texttt{ or } (a_3 \texttt{ = } a_1)$$

The formula uses only monotone constructs and is therefore a monotone formula. Any binding which maps the three variables a_1, a_2, and a_3 to three distinct atoms provides a counterexample to the formula since each of the disjuncts will be false. However, any binding within a scope of 2 must map two of the three variables to the same atom, making one of the disjuncts, and hence the overall formula, true.

□

The disjunction operator is needed to model strongly-data-independent systems, where non-determinism is present e.g. a system with two inputs which non-deterministically selects one and copies it to its output. However we have shown that the use of disjunction, although a monotone operator, gives the possibility of formulas which are not verifiable within a scope of 2 and hence not ones which describe strongly-data-independent systems.

Summary

In this chapter we have considered a restricted form of data independence, strong data independence, where the operation of equality testing variables of the data-independent type variable is not permitted. Based on intuition and other work involving strong data independence we expect a scope of 2 to be sufficient to check strongly-data-independent system in Alloy.

To explore why this was a difficult task, we introduced an even smaller language than the one used in the thesis so far. We defined the notion of monotone formulas and monotone constructors and saw which of the constructors of the smaller language were monotone. We explained why formulas describing strongly-data-independent systems are monotone. However, we found that a scope of 2 was insufficient to check monotone formulas in general and therefore the monotonicity property was not sufficient to describe formulas corresponding to strongly-data-independent systems.

We conclude that the procedure to decide whether a formula describes a strongly-data-independent system (if such a procedure exists) is not simply a matter of inspecting which language constructors it uses and must be more complex.

In Alloy and the small language of this chapter, the same syntactic construct is used to model both variable assignment and equality testing, and it may be that this makes it difficult to capture the notion of strong data independence at a syntactic level. In other formalisms where a theory of strong data independence has been developed e.g. CSP different constructs are used.

Chapter 9

Case study

In this chapter we present an example problem and show how the small model theorem (in particular the second version i.e. Theorem 7.17) can be applied to generate thresholds on data independent type variables. The example is taken from [Jac06, Section 6.2] and concerns a system used for hotel room locking.

The problem is described using the full-blown Alloy language, which although similar to the language presented in Chapter 2 has some differences. In particular we now use Convention 2.21, concerning default multiplicity keywords, which we have so far avoided.

The reader may not be familiar with the full-blown Alloy language but we supplement the formal description with a commentary, and so a complete understanding is not necessary. We refer the interested reader to [Jac06] for a complete description of the Alloy language.

Note that since the theorems in the thesis apply to the language in Chapter 2, not the full-blown Alloy language, some translation of the example problem is required before the theorems can be applied. In this chapter we leave this translation implicit. As we mentioned in Chapter 2 such a translation is made by the Alloy Analyzer tool, so it would be relatively easy to use the results of the thesis within this tool.

9.1 Hotel room locking

Most hotels issue disposable room keys; when you check out you can take your key with you. How then can the hotel prevent your from reentering your room after it has been assigned to someone else?

The trick is *recodable* locks. When the hotel issues a new key to the next occupant and they insert this in the lock, this recodes the lock, so that the previous key will no longer work. The beauty of this scheme is that it requires no communication between the hotel front desk and the door locks, so long as they are synchronized initially.

A model of the hotel room locking system is defined using the full-blown Alloy language, which begins:

```
1  module hotel
2  open util/ordering[Time] as to
3  open util/ordering[Key] as ko
4  sig Key, Time {}
5  sig Room {
6      keys: set Key,
7      currentKey: keys one -> Time
8      }
9  one sig FrontDesk {
10     lastKey: (Room -> lone Key) -> Time,
11    occupant: (Room -> lone Guest) -> Time
12     }
13 sig Guest {
14     keys: Key -> Time
15     }
```

Line 4 declares the type variables `Key` and `Time`. Adding a `Time` atom to a relation makes it time dependent. `Key` is used to model the hotel room keys. An ordering is given to these types by importing the module `util/ordering` on Lines 2-3. It defines functions: `to/first()`, `to/last()`, `to/next(t:Time)` on `Time`. Likewise, on `Key`, `ko/next()` etc. is defined. An ordering is given to `Key` to represent key generation.

Lines 5-8 define the type variable `Room` and some relations with domain `Room`. (In Alloy parlance, these relations are known the *fields* of `Room`.) The relation `keys` has range `set Key` and represents the set of keys used for a particular room. The relation `currentKey` has range `keys one -> Time` and represents for a particular room and time the key that will open it.

The front desk is modeled with the `FrontDesk` type variable which is defined to be a singleton on line 9 with the keyword `one`. `lastKey` represents the front desk's record of the last key issued for each room. `occupant` represents the front desk's view of who is allowed in each room.

Lines 13-15 define a type variable `Guest` to represent guests, and a relation `keys` to represent which keys they hold. (Note that `keys` is actually the union of two relations since it was also defined on line 6).

```
16 fact DisjointKeySets {
17    Room <: keys : Room lone -> Key
18     }
19 fun nextKey (k: Key, ks: set Key): set Key {
20     ko/min (ko/nexts (k) & ks)
21    }
```

`DisjointKeySets` is a fact of the model which asserts that each key opens a maximum of one room. `<:` is the domain restriction operator. Facts of the model are conjoined with any formulas being checked for consistency or validity.

`nextKey` is a function, which models the generation of the successor key for a room given its current key `k` and its set of keys `ks`.

```
22 pred init (t: Time) {
23     no Guest.keys.t
24     no FrontDesk.occupant.t
25     all r: Room | FrontDesk.lastKey.t [r] = r.currentKey.t
26     }
```

The above predicate will be used to assert the following. Initially guests hold no keys. According to the front desk no room is occupied. The front desk has a record of which key was last given for which room, which is up to date with which key currently opens which room. The square brackets used on line 26 are another form of the join operator: `a[b]` is equivalent to `b.a`.

```
27 pred checkin (t, t': Time, g: Guest, r: Room, k: Key) {
28     g.keys.t' = g.keys.t + k
29     let occ = FrontDesk.occupant {
30         no occ.t [r]
31         occ.t' = occ.t + r -> g
32         }
33     let lk = FrontDesk.lastKey {
34         lk.t' = lk.t ++ r -> k
35         k = nextKey (lk.t [r], r.keys)
36         }
37     noRoomChangeExcept (t, t', none)
38     noGuestChangeExcept (t, t', g)
39     }
```

The above predicate models the action of a guest checking in. When a guest checks in for a room they are given the the next key in the set of keys for that room. The front desk records must indicate that the room is unoccupied. The front desk updates its records of who occupies the room and which key was last given for the room.

```
40 pred entry (t, t': Time, g: Guest, r: Room, k: Key) {
41     k in g.keys.t
42     let ck = r.currentKey |
43         (k = ck.t and ck.t' = ck.t) or
44         (k = nextKey(ck.t, r.keys) and ck.t' = k)
45     noRoomChangeExcept (t, t', r)
46     noGuestChangeExcept (t, t', none)
47     noFrontDeskChange (t, t')
48     }
```

The above predicate models a guest entering a room. When a guest enters a room with a key, they must hold that key. The key can be either the current key

for that room, or the next key in the set of keys for that room. In the former case the current key does not change, but in the latter the current key becomes the next key in the set of keys for that room.

```
49 pred checkout (t, t': Time, g: Guest) {
50     let occ = FrontDesk.occupant {
51         some occ.t.g
52         occ.t' = occ.t - Room->g
53         }
54     FrontDesk.lastKey.t = FrontDesk.lastKey.t'
55     noRoomChangeExcept (t, t', none)
56     noGuestChangeExcept (t, t', none)
57     }
```

The above predicate models a guest checking out. When a guest checks out the front desk updates its records to indicate that the guest occupies no rooms.

```
58 pred noFrontDeskChange (t, t': Time) {
59     FrontDesk.lastKey.t = FrontDesk.lastKey.t'
60     FrontDesk.occupant.t = FrontDesk.occupant.t'
61     }
62 pred noRoomChangeExcept (t, t': Time, rs: set Room) {
63     all r: Room - rs | r.currentKey.t = r.currentKey.t'
64     }
65 pred noGuestChangeExcept (t, t': Time, gs: set Guest) {
66     all g: Guest - gs | g.keys.t = g.keys.t'
67     }
```

To ensure that nothing else happens during the events modeled, the above predicates are used.

```
68 fact Traces {
69     init (to/first ())
70     all t: Time - to/last() | let t' = to/next (t) |
71         some g: Guest, r: Room, k: Key |
72             entry (t, t', g, r, k)
73             or checkin (t, t', g, r, k)
74             or checkout (t, t', g)
75 }
76 assert NoBadEntry {
77     all t: Time, r: Room, g: Guest, k: Key |
78         let t' = to/next(t) |
79             let o = FrontDesk.occupant.t [r] |
80                 entry (t, t', g, r, k) and some o => g in o
81 }
82 fact NoIntervening {
83     all t: Time - to/last() |
```

```
84          let t'=to/next (t), t'' = to/next(t') |
84              all g: Guest, r:Room, k:Key |
85                  checkin (t, t', g, r, k) =>
86                      ( entry (t', t'', g, r, k) or no t'')
87      }
88 check NoBadEntry for 3 but 2 Room, 2 Guest, 5 Time
```

The first fact above ensures that the model is correctly initialized and that events associated with the model occur at each time step.

The `NoBadEntry` assertion states that if a guest enters a room, and the front desk records show the room is occupied, then the front desk records show that the room is occupied by that guest. An assertion of the model is checked by the Alloy Analyzer (in conjunction with the facts of the model).

When `NoBadEntry` is checked without the fact `NoIntervening` a counterexample is found. This is because a guest can check in for a room, enter that room and then check out. If another guest checks in for the same room, then until they go to their room, the first guest can enter their room since the room lock has not yet been recoded.

The `NoIntervening` fact is added to ensure the next event after a guest checks in for a room is for them to enter that room. With this fact added, no counterexamples to `NoBadEntry` are found.

The last line sets the scope of the check which is 3 for all type variables except 2 for `Room` and `Guest` and 5 for `Time`. Even though no counterexamples are found at this scope the question remains as to whether counterexamples could be found at a larger scope. Some of the type variables in the problem are data independent and therefore we can remove the guesswork for these types by applying our small model theorem.

9.1.1 Translation to thesis language

The problem presented above can be translated to the language of the thesis (given in Chapter 2). We have already explained how the *fields* of type variables correspond to variables which stand for relations. The definitions of predicates and functions can be expanded and placed in-line. The more esoteric operators of the full-blown language such as override (used on line 34) can be rewritten using the operators of the thesis language. This is a task best left to a machine and we continue working at the level of abstraction of the full-blown language as this is more illuminating for the reader; an expansion of the problem into the thesis language would be overwhelming.

The only difficult aspect of translation are lines 7, where an expression, `keys`, is used in declaring the type of the field `currentKey`, and line 17 where it is asserted that an expression is of a particular type. There seems to be a range of possibilities of how to translate these constructs and we are not aware of the approach used by the Alloy Analyzer.

An optimal way to eliminate these constructs is one which does not introduce variables unnecessarily as these would add to thresholds. One such way proceeds

by replacing the previous definitions of `Key` and `Room` and the `DisjointKeys` fact with:

```
sig Key {
    room : lone Room
}
sig Room {
    currentKey : Key one -> Time
}
fact RoomKeys {
     currentKey.Time in ~room
}
```

The effect of this is that the `keys` field of the `Room` type variable (line 6) is replaced with the `room` field of the `Key` of the type variable, which represents its transpose. The `DisjointKeys` fact is no longer needed due to the multiplicity constraint on `room`. The `RoomKeys` fact is used to eliminate the use of an expression in the declaration of the type of `currentKey`.

A consequence of these changes and that `keys` is still in use on line 14 would be that all other references to `keys` in the model should be replaced by `keys + ~room`.

9.1.2 Data independent type variables

The model introduces five base types: `Key`, `Time`, `Room`, `FrontDesk` and `Guest`.

`FrontDesk` is defined to be a singleton type by the keyword `one`, i.e. the type `FrontDesk` has exactly one element and a scope of 1 for `FrontDesk` is sufficient. There is no point attempting to apply the small model theorem to `FrontDesk`.

To define an ordering on a type, a relation between that type and itself is declared in the module `util/ordering`. The contribution of such relations to thresholds is infinite, so there is no point in attempting to apply the small model theorem to `Time` or `Key`.

The types `Guest` and `Room` are left, and both are candidates on which to apply the small model theorem. In this subsection we shall derive a threshold for `Guest`. A threshold for `Room` would also be possible.

Negation normal form

Recall that the use of quantification over `Guest` is not a data independent construct. We need need to eliminate such quantification from the formula and the first step is to put the formula in negation normal form (with respect to quantifiers over `Guest`. i.e. push negation inside any `Guest` quantifiers).

We start be substituting the `NoBadEntry` assertion, for its negation which we call `BadEntry`

```
pred BadEntry() {
    some t: Time, r: Room, g: Guest, k: Key |
```

```
        let t' = to/next(t) |
            let o = FrontDesk.occupant.t [r] |
                !(entry (t, t', g, r, k) and some o => g in o)
}
```

and changing the `check` command to a `run` command, which looks for instances of the formula rather than counterexamples:

```
run BadEntry for 3 but 2 Room, 2 Guest, 5 Time
```

Rewrite universal quantification and set comprehension

Whereas existential quantification over `Guest` can be eliminated automatically using Skolemization, universal quantification and set comprehension over `Guest` has to be removed manually.

We rewrite:

```
pred noGuestChangeExcept (t, t': Time, gs: set Guest) {
    all g: Guest - gs | g.(keys).t = g.(keys).t'
    }
```

as:

```
pred noGuestChangeExcept (t, t': Time, gs: set Guest) {
    ((Guest - gs) -> Key) & keys.t =
        ((Guest - gs) -> Key) & keys.t'
    }
```

Recall that the analysis tool conjoins the facts of the model: `Traces` and `NoIntervening` with the `BadEntry` assertion and looks for a model of the resultant formula.

The `NoIntervening` fact still uses universal quantification. However we can see from the `Traces` fact that only one guest can check in at any time. By combining these two facts we can eliminate universal quantification. That is we replace:

```
fact Traces {
    init (to/first ())
    all t: Time - to/last() | let t' = to/next (t) |
        some g: Guest, r: Room, k: Key |
            entry (t, t', g, r, k)
            or checkin (t, t, g, r, k)
            or checkout (t, t, g)
}
```

and:

```
fact NoIntervening {
    all t: Time - to/last() |
```

```
        let t'=to/next (t), t'' = to/next(t') |
            all g: Guest, r:Room, k:Key |
                checkin (t, t', g, r, k) =>
                    (entry (t', t'', g, r, k) or no t'')
    }
```

with:

```
fact TracesAndNoIntervening {
    init (to/first ())
    all t: Time - to/last() |
        let t' = to/next (t), t'' = to/next(t')|
            some g: Guest, r: Room, k: Key |
                entry (t, t', g, r, k)
                or (checkin (t, t', g, r, k)
                    and (entry (t', t'', g, r, k) or no t''))
                or checkout (t, t', g)
}
```

Skolemize

We introduce Skolem variables. We replace:

```
sig Key, Time {}
```

with:

```
sig Key {}
sig Time {
    g: Guest
    }
```

So that we can rewrite:

```
fact TracesAndNoIntervening {
    init (to/first ())
    all t: Time - to/last() |
        let t' = to/next (t), t'' = to/next(t')|
            some g: Guest, r: Room, k: Key |
                entry (t, t', g, r, k)
                or (checkin (t, t', g, r, k)
                    and (entry (t', t'', g, r, k) or no t''))
                or checkout (t, t', g)
}
```

as:

```
fact TracesAndNoIntervening {
    init (to/first ())
```

```
    all t: Time - to/last() |
        let t' = to/next (t), t'' = to/next(t')|
            some r: Room, k: Key |
                entry (t, t', t.g, r, k)
                or (checkin (t, t', t.g, r, k)
                    and (entry (t', t'', t.g, r, k) or no t''))
                or checkout (t, t', t.g)
}
```

We also add:

```
one sig BadGuest in Guest {}
```

which introduces a Skolem variable of type `one Guest` called `BadGuest`, so that we can rewrite:

```
pred BadEntry() {
    some t: Time, r: Room, g: Guest, k: Key | let t' = to/next(t) |
        let o = FrontDesk.occupant.t [r] |
            !(entry (t, t', g, r, k) and some o => g in o)
}
```

as:

```
pred BadEntry() {
    some t: Time, r: Room, k: Key | let t' = to/next(t) |
        let o = FrontDesk.occupant.t [r] |
            !(entry (t, t', BadGuest, r, k) and some o => BadGuest in o)
}
```

We have now eliminated all quantification over `Guest`.

Calculate Threshold

Recall that line 88 of the model specifies the scope of the check of the `NoBadEntry` assertion:

```
88 check NoBadEntry for 3 but 2 Room, 2 Guest, 5 Time
```

By using the small model theorem, with `Guest` as the data-independent type variable, we can check the assertion within the following scope:

$$\begin{aligned}\Theta = \{\texttt{Time} &\mapsto 5\\ \texttt{Room} &\mapsto 2\\ \texttt{Key} &\mapsto 2\\ \texttt{FrontDesk} &\mapsto 1\\ \texttt{Guest} &\mapsto \infty\}\end{aligned}$$

To calculate a threshold for the scope of `Guest` we need to calculate the contribution of the variables which use `Guest`. Having made various transformations to the model and introduced Skolem variables, we give a recap of the current set of variables used by the formula.

```
sig Time {
    g: Guest
    }
sig Key {
    room : lone Room
}
sig Room {
    currentKey : Key one -> Time
}
one sig FrontDesk {
    lastKey: (Room -> lone Key) -> Time,
    occupant: (Room -> lone Guest) -> Time
    }
sig Guest {
    keys: Key -> Time
    }
one sig BadGuest in Guest
```

The contribution of `g` is:

$$
\begin{aligned}
&\mathit{Bound}_{\texttt{Guest},\Theta}(\texttt{Time -> one Guest}) \\
&= \\
&\mathit{RelBound}_{\texttt{Guest},\Theta}(\texttt{Time -> one Guest}) \\
&= \\
&(\mathit{Columns}_{\texttt{Guest}}(\texttt{Guest}) * \mathit{Size}_{\Theta}(\texttt{Time}), 1) \\
&= \\
&(1 * \Theta(\texttt{Time}), 1) \\
&= \\
&(5, 1)
\end{aligned}
$$

The contribution of `occupant` is:

$$
\begin{aligned}
&\mathit{Bound}_{\texttt{Guest},\Theta}(\texttt{FrontDesk -> ((Room -> lone Guest) -> Time)}\\
&=\\
&\mathit{RelBound}_{\texttt{Guest},\Theta}(\texttt{FrontDesk -> ((Room -> lone Guest) -> Time)}\\
&=\\
&\mathit{Size}_{\Theta}(\texttt{FrontDesk}) * \mathit{RelBound}_{\texttt{Guest},\Theta}(\texttt{(Room -> lone Guest) -> Time})\\
&=\\
&\Theta(\texttt{FrontDesk}) * \mathit{Size}_{\Theta}(\texttt{Room}) * \mathit{RelBound}_{\texttt{Guest},\Theta}(\texttt{Room -> lone Guest})\\
&=\\
&1 * \Theta(\texttt{Room}) * (\mathit{Columns}_{\texttt{Guest}}(\texttt{Guest}) * \mathit{Size}_{\Theta}(\texttt{Room}), 1)\\
&=\\
&5 * (1 * \Theta(\texttt{Room}), 1)\\
&=\\
&5 * (2, 1)\\
&=\\
&(10, 1)
\end{aligned}
$$

The contribution of `keys` can be ignored. This is because it has a malleable type and every use of `keys` in the above problem is as the right argument of a join.

Finally the Skolem variable `BadGuest` of type `one Guest` has a contribution of $(1, 1)$. The total contribution is therefore $(5, 1) * (10, 1) * (1, 1) = (16, 1)$.

We now invoke Theorem 7.17 which tells us a scope of $16 + 1 = 17$ is a threshold scope for `Guest`. We change scope of the `run` command to:

```
run BadEntry for 3 but 2 Room, 17 Guest, 5 Time
```

and run the analyzer which finds no instance. This analysis is complete with respect to `Guest`. We can be certain that increasing the scope on `Guest` beyond 17 is fruitless.

Chapter 10

Conclusions and further work

10.1 Conclusions

In Section 1.2 we set out the goal of the thesis, namely to give a small model theorem which applies to data-independent systems in Alloy.

In Chapter 2 we defined the modeling language used for the small model theorem, based on the Alloy Kernel language. We defined the notion of scope and explained what it means for a formula to be valid or consistent within a particular scope.

In Chapter 3 we laid the foundations for the small model theorem by giving a semantic definition of data independence. To do this, we defined a set of quotient atoms to be used as a basis for the interpretation of formulas instead of the usual set of atoms. The quotient atoms were formed by the projection of an equivalence relation on the usual set of atoms. By lifting this projection we obtained a Galois insertion between the usual set of values used to bind values and interpret expressions, and a more abstract set of quotient values, thereby obtaining a means to compare the usual and more abstract interpretations. We then defined a formula to be semantically data independent, when its truth value under a particular binding was always the same as its truth value under the corresponding quotient binding, for any appropriate choice of equivalence relation. Similarly we defined expressions to be semantically data independent, except rather than seeking the values under both interpretations to be identical we ask them to be comparable using the Galois insertion.

In Chapter 4 we observed that we had no means to automatically decide whether a formula satisfied the semantic definition of data-independence and set about giving a syntactic definition, against which a formula could be tested automatically. We started by introducing the notion of data-independent language constructors and investigated which language constructors were data-independent and which were not. We then defined a formula to be syntactically

data independent if it only used data-independent constructors. By using an induction over the syntax of such formulas, we proved that that the syntactic definition implied the semantic one.

Having obtained a means to establish when it is sound to use the more abstract interpretation, we set out in Chapter 5 to calculate a bound on the scope of the data-independent type variable when using this abstract interpretation. We started by defining the maximum data relation for a value and the maximum data relation for a binding. The latter is the coarsest equivalence relation on the original set of atoms that one can use to generate a set of quotient atoms which reliably interpret a data-independent formula, and thus give rise to an interpretation of the data-independent type variable with the fewest elements. We then showed how we could obtain an upper bound on this number of elements as a function of the type of every variable in the binding. We then gave an explicit calculation of the upper bound, and proved a small model theorem: if a data-independent formulas has any model, it has a model whose scope is within this upper bound. The final part of this chapter was spent giving some example formulas which have models at the calculated threshold scope, but no models at smaller scopes, thereby indicating that the bounds we had derived could not be made smaller.

The following chapter gave some techniques to rewrite formulas which consequently either make the small model theorem applicable when it otherwise would not be, or reduce the size of threshold that the theorem generates.

In Chapter 7 we looked a formulas with multiple data-independent types. We explained how the small model theorem from Chapter 5 could in some cases be applied repeatedly in order to generate thresholds scopes for multiple data-independent type variables, but that this technique of repeated application of the small model theorem broke down when the formula in question has a variable which is a relation between the data-independent types. We overcame this limitation by modifying the set of quotient atoms used in the small model theorem to give two more small model theorems, applicable in different circumstances. The first of these we proved but the second we actually left as a conjecture since it would require much rework of the preceding chapters.

In Chapter 8 we considered a special case of data independence called strong data independence, where the operation of equality testing between values of the data-independent type is no longer permitted. Based on intuition and other work involving strong data independence we wanted to show that a scope of 2 would be sufficient for a strongly-data-independent type variable. This was not achieved and instead we explored why this was a difficult task. To do this we defined an even smaller language than the one used in the thesis so far. We then defined the concept of monotone language constructors and explained why formulas describing strongly-data-independent systems use only monotone constructors. We then gave an example of some formulas which use only such constructors and which have models at a scope of 3 but no models at a scope of 2. This indicated that the monotonicity property was insufficient to describe formulas corresponding to strongly-data-independent formulas. We concluded that the procedure to determine whether a formula describes a strongly-data-

independent was not simply a matter of inspecting which language constructors it uses.

Finally, in Chapter 9 we gave an example of how to apply the small model theorem to an Alloy model of disposable room keys issued by hotels.

In summary we conclude that we have given a small model theorem as we set out to do, and that this theorem completes the Alloy model finder's analysis of formulas with respect to data-independent type variables.

10.2 Further work

We suggest that further work could explore the following avenues.

10.2.1 Multiple data-independent type variables

The theory given in Chapter 7 shows that it is possible in some circumstances to complete the analysis of a formula with respect to multiple data-independent type variables. A conjecture was given in the chapter, which if proved correct, would expand the range of circumstances in which analysis can be completed.

The theory is also limited by the condition that every relation involving multiple data-independent type variables must first be used in a join over the data-independent type variable when they appear in formulas. Some such condition is necessary, but this one does not allow for example relations `W -> X -> Y -> Z` where `X` and `Y` are data-independent type variables but `W` and `Z` are not.

10.2.2 Strong data independence

Although the small model theorem presented in the thesis is applicable to strongly-data-independent systems it will generally produce much greater thresholds than necessary which may lead to state explosion. In Chapter 8 we looked at strong data independence and explained some of the difficulties in developing a small model theorem specifically for strongly-data-independent systems. Overcoming these difficulties is an area for further research.

10.2.3 Implementation within the Alloy Analyzer

As we mentioned in our conclusions the small model theorems in the thesis can be applied automatically and this could be implemented within the Alloy Analyzer.

Statements could be added to the Alloy language to direct the Analyzer to check that models are data independent with respect to certain type variables, and if so calculate appropriate thresholds. Furthermore, the possible failure of such checks would provide a further means to detect errors in models.

Appendix A

Proofs

This appendix contains some additional proofs.

Theorem A.1
Let T be a type expression. Let δ be a set map such that:

$$\mathit{Used}(T) \subseteq \operatorname{dom} \delta$$

Let $\sim : \mathit{DataRel}(\delta)$. Then:

$$([T]_\delta)/\!\sim = [T]_{\delta/\sim}$$

$\diamond$

Proof The proof uses structural induction over T.
Case X.

$$\begin{array}{ll}
& ([X]_\delta)/\!\sim \\
= & \qquad [2.10] \\
& \{x : \delta(X) \bullet \langle x \rangle\}/\!\sim \\
= & \qquad [3.8] \\
& \{x : \delta(X) \bullet \langle x \rangle/\!\sim\} \\
= & \qquad [3.8] \\
& \{x : \delta(X) \bullet \langle [x]_\sim \rangle\} \\
= & \qquad [3.19] \\
& \{x' : (\delta/\!\sim)(X) \bullet \langle x' \rangle\} \\
= & \qquad [2.10] \\
& [X]_{\delta/\sim}
\end{array}$$

Case P + Q

$$
\begin{array}{ll}
& ([P + Q]_\delta)/\sim \\
= & \qquad [2.10] \\
& ([P]_\delta \cup [Q]_\delta)/\sim \\
= & \qquad [3.8] \\
& (([P]_\delta)/\sim) \cup (([Q]_\delta)/\sim) \\
= & \qquad [\text{ind. hyp.}] \\
& [P]_{\delta/\sim} \cup [Q]_{\delta/\sim} \\
= & \qquad [2.10] \\
& [P + Q]_{\delta/\sim}
\end{array}
$$

Case P -> Q.

$$
\begin{array}{ll}
& ([P \text{ -> } Q]_\delta)/\sim \\
= & \qquad [2.10] \\
& \{xs : [P]_\delta;\ ys : [Q]_\delta \bullet xs \frown ys\}/\sim \\
= & \qquad [3.8] \\
& \{xs : [P]_\delta;\ ys : [Q]_\delta \bullet (xs \frown ys)/\sim\} \\
= & \qquad [3.8] \\
& \{xs : [P]_\delta;\ ys : [Q]_\delta \bullet (xs/\sim) \frown (ys/\sim)\} \\
= & \qquad [3.8] \\
& \{xs' : ([P]_\delta)/\sim;\ ys' : ([Q]_\delta)/\sim \bullet xs' \frown ys'\} \\
= & \qquad [\text{ind. hyp.}] \\
& \{xs' : [P]_{\delta/\sim};\ ys' : [Q]_{\delta/\sim} \bullet xs' \frown ys'\} \\
= & \qquad [2.10] \\
& [P \text{ -> } Q]_{\delta/\sim}
\end{array}
$$

Case $m\ T$.

$$
\begin{array}{ll}
& ([m\ T]_\delta)/\sim \\
= & \qquad [2.10] \\
& ([T]_\delta)/\sim \\
= & \qquad [\text{ind. hyp.}] \\
& [T]_{\delta/\sim} \\
= & \qquad [2.10] \\
& [m\ T]_{\delta/\sim}
\end{array}
$$

Case T m->n U.
The proof is entirely similar to the case P -> Q. □

Lemma A.2 If a value has a multiplicity then so does its quotient.

Let V be a value and $\sim$ be an equivalence relation on atoms. Let m be a multiplicity keyword. It follows that:

$$[\![m]\!](V) \Rightarrow [\![m]\!](V/\sim)$$

$\diamond$

Proof

$$\begin{array}{ll} & \#V = 1 \\ \Rightarrow & [2.2] \\ & (\exists\, xs : \mathrm{seq}\, Atom \bullet V = \{xs\}) \\ \Rightarrow & [3.8] \\ & (\exists\, xs : \mathrm{seq}\, Atom \bullet V/\sim = \{xs/\sim\}) \\ \Rightarrow & \\ & \#(V/\sim) = 1 \end{array}$$

With reference to 2.14, this proves the case $m =$ `one`.

$$\begin{array}{ll} & \#V = 0 \\ \Leftrightarrow & \\ & V = \emptyset \\ \Leftrightarrow & [3.8] \\ & V/\sim = \emptyset \\ \Leftrightarrow & \\ & \#(V/\sim) = 0 \end{array}$$

With reference to 2.14, this proves the case $m =$ `some` by noting the condition $\#V \geq 1$ is equivalent to $\neg\, \#V = 0$. It also prove the case $m =$ `lone` by noting the condition $\#V \leq 1$ is equivalent to $\#V = 1 \vee \#V = 0$. The case $m =$ `set` is trivially true. $\square$

Definition A.3 Equivalent sequences

An equivalence relation on atoms, $\sim$, induces an equivalence relation on sequences of atoms given by:

$$xs \sim ys \Leftrightarrow \#(xs) = \#(ys) \wedge (\forall\, i : 1 \,..\, \#(xs) \bullet xs(i) \sim ys(i))$$

$\diamond$

Lemma A.4 Concatenated equivalents

Let $xs_1, xs_2, ys_1, ys_2 : \mathrm{seq}\, Atom$. Suppose:

$$xs_1 \sim xs_2 \wedge ys_1 \sim ys_2$$

Then:

$$xs_1 \frown ys_1 \sim xs_2 \frown ys_2$$

$\diamond$

Proof

$$\begin{array}{ll} & xs_1 \sim xs_2 \wedge ys_1 \sim ys_2 \\ \Rightarrow & \qquad [\text{A.3}] \\ & \#(xs_1) = \#(xs_2) \wedge \#(ys_1) = \#(ys_2) \\ \Rightarrow & \\ & \#(xs_1 \frown ys_1) = \#(xs_1) + \#(ys_1) = \#(xs_2) + \#(ys_2) = \#(xs_2 \frown ys_2) \end{array}$$

Let $i : 1 \mathinner{..} \#(xs_1 \frown ys_1)$.
Case $i : 1 \mathinner{..} \#(xs_1)$.

$$(xs_1 \frown ys_1)(i) = xs_1(i) = xs_2(i) = (xs_2 \frown ys_2)(i)$$

Case $i : \#(xs_1) + 1 \mathinner{..} \#(xs_1 \frown ys_1)$.

$$\begin{array}{l} (xs_1 \frown ys_1)(i) = ys_1(i - \#(xs_1)) = \\ \qquad ys_2(i - \#(xs_1)) = ys_2(i - \#(xs_2)) = (xs_2 \frown ys_2)(i) \end{array}$$

Hence:

$$\forall\, i : 1 \mathinner{..} \#(xs_1 \frown ys_1) \bullet (xs_1 \frown ys_1)(i) = (xs_2 \frown ys_2)(i)$$

And thus:

$$xs_1 \frown ys_1 \sim xs_2 \frown ys_2$$

$\square$

Lemma A.5 Sequences with identical quotients are equivalent
Let $xs, ys : \mathrm{seq}\, Atom$ and $\sim : Eqrel$. Then:

$$xs/\!\sim = ys/\!\sim \Leftrightarrow xs \sim ys$$

$\diamond$

Proof If $\#(xs) \neq \#(ys)$ then $xs/\!\sim \neq ys/\!\sim$ and $\neg\ xs \sim ys$ so both sides of the above logical equivalence are false.

It is then sufficient to consider cases where $\#(xs) = \#(ys)$. The proof uses induction on the length of the sequences.

Case $xs = \langle\rangle$.

It follows $ys = \langle\rangle$ and the lemma holds vacuously.

Case $xs = \langle x \rangle \frown xs_1$ and $ys = \langle y \rangle \frown ys_1$.

$$
\begin{array}{ll}
(\langle x \rangle \frown xs_1)/\!\sim = (\langle y \rangle \frown ys_1)/\!\sim & \\
\Leftrightarrow & [3.8] \\
(\langle x \rangle/\!\sim) \frown (xs_1/\!\sim) = (\langle y \rangle/\!\sim) \frown (ys_1/\!\sim) & \\
\Leftrightarrow & [3.8] \\
\langle [x]_\sim \rangle \frown (xs_1/\!\sim) = \langle [y]_\sim \rangle \frown (ys_1/\!\sim) & \\
\Leftrightarrow & \\
[x]_\sim = [y]_\sim \wedge xs_1/\!\sim = ys_1/\!\sim & \\
\Leftrightarrow & [\text{ind. hyp.}] \\
x \sim y \wedge xs_1 \sim ys_1 & \\
\Leftrightarrow & [\text{A.3}] \\
\langle x \rangle \sim \langle y \rangle \wedge xs_1 \sim ys_1 & \\
\Leftrightarrow & [\text{A.3, A.4}] \\
\langle x \rangle \frown xs_1 \sim \langle y \rangle \frown ys_1 &
\end{array}
$$

□

Definition A.6 Alternative definition of closed
Let V be a value and $\sim$ an equivalence relation on atoms. We define:

$$
\begin{array}{l}
Closed^2_\sim(V) \Leftrightarrow \\
\quad (\forall\, xs : \mathrm{seq}\, Atom;\ ys : \mathrm{seq}\, Atom \mid xs \sim ys \bullet xs \in V \Rightarrow ys \in V)
\end{array}
$$

◇

Theorem A.7 Both definitions of closed are equivalent
Let V be a value and $\sim$ an equivalence relation on atoms. Then:

$$Closed_\sim(V) \Leftrightarrow Closed^2_\sim(V)$$

◇

Proof Suppose $Closed_\sim(V)$. Let $xs;\ ys : \mathrm{seq}\, Atom$ such that $xs \sim ys$.

$$
\begin{array}{ll}
xs \in V & \\
\Rightarrow & [3.8] \\
xs/\!\sim \in V/\!\sim & \\
\Rightarrow & [\text{A.5}] \\
ys/\!\sim = xs/\!\sim \in V/\!\sim & \\
\Rightarrow & [3.9, 3.11] \\
ys \in (ys/\!\sim).\sim \subseteq (V/\!\sim).\sim & \\
\Rightarrow & [3.16]
\end{array}
$$

$$ys \in (V/\sim).\sim = V$$

This shows $Closed^2_\sim(V)$

Now suppose $Closed^2_\sim(V)$. 3.13 gives us $V \subseteq (V/\sim).\sim$. To complete the proof we must show: $(V/\sim).\sim \subseteq V$. So now let $xs : (V/\sim).\sim$. Choose $xs' : V/\sim$ such that $xs \in xs'.\sim$. Then choose $ys : V$ such that $xs' = ys/\sim$. It follows:

$$\begin{array}{ll}
& xs \in xs'.\sim \\
\Rightarrow & [3.8,\ 3.12] \\
& xs/\sim \in (xs'.\sim)/\sim = \{xs'\} \\
\Rightarrow & \\
& xs/\sim = xs' = ys/\sim \\
\Rightarrow & [A.5] \\
& xs \sim ys \in V \\
\Rightarrow & [A.6] \\
& xs \in V
\end{array}$$

□

Theorem A.8

Let T be a type expression. Let δ be a set map such that:

$$Used(T) \subseteq \operatorname{dom} \delta$$

Let $V : [\![T]\!]_\delta$. Let $\sim$ be a data relation. Suppose $Closed_\sim(V)$. Then:

$$V/\sim \in [\![T]\!]_{\delta/\sim}$$

◇

Proof The proof uses structural induction.

Case P ($P \in FixedArityType$).

$$\begin{array}{ll}
& V \in [\![P]\!]_\delta \\
\Rightarrow & [2.18] \\
& V \in \mathbb{P}([P]_\delta) \\
\Rightarrow & \\
& (\forall\, zs : V \bullet zs \in [P]_\delta) \\
\Rightarrow & [3.8] \\
& (\forall\, zs : V \bullet zs/\sim \in ([P]_\delta)/\sim) \\
\Rightarrow & [3.8] \\
& (\forall\, zs' : V/\sim \bullet zs' \in ([P]_\delta)/\sim) \\
\Rightarrow & [A.1]
\end{array}$$

$$
\begin{array}{ll}
& (\forall\, zs' : V/\!\sim \bullet\ zs' \in [P]_{\delta/\sim}) \\
\Rightarrow & \\
& V/\!\sim \in \mathbb{P}([P]_{\delta/\sim}) \\
\Rightarrow & [2.18] \\
& V/\!\sim \in [\![P]\!]_{\delta/\sim}
\end{array}
$$

Case $m\ T$.

$$
\begin{array}{ll}
& V \in [\![m\ T]\!]_\delta \\
\Rightarrow & [2.18] \\
& V \in [\![T]\!]_\delta \wedge [\![m]\!](V) \\
\Rightarrow & [\text{ind.hyp., A.2}] \\
& V/\!\sim \in [\![T]\!]_{\delta/\sim} \wedge [\![m]\!](V/\!\sim) \\
\Rightarrow & [2.18] \\
& V/\!\sim \in [\![m\ T]\!]_{\delta/\sim}
\end{array}
$$

Case $T\ m\texttt{->}n\ U$

$$V \in [\![T\ m\texttt{->}n\ U]\!]_\delta$$

By 2.18, one can choose $rel : [T]_\delta \leftrightarrow [U]_\delta$ such that[1] :

$$
\begin{array}{l}
(\forall\, xs : [T]_\delta \bullet rel(\!|\ \{xs\}\ |\!) \in [\![n\ U]\!]_\delta) \wedge \\
(\forall\, ys : [U]_\delta \bullet rel^{\sim}(\!|\ \{ys\}\ |\!) \in [\![m\ T]\!]_\delta) \wedge \\
V = \{pair : rel \bullet \mathit{first}(pair) \frown \mathit{second}(pair)\}
\end{array}
$$

Then let:

$$rel' = \{pair : rel \bullet (\mathit{first}(pair)/\!\sim, \mathit{second}(pair)/\!\sim)\}$$

It follows:

$$
\begin{array}{ll}
& rel' \in (([T]_\delta)/\!\sim) \leftrightarrow (([U]_\delta)/\!\sim) \\
\Rightarrow & [A.1] \\
& rel' \in ([T]_{\delta/\sim}) \leftrightarrow ([T]_{\delta/\sim})
\end{array}
$$

And:

$$
\begin{array}{ll}
& V = \{pair : rel \bullet \mathit{first}(pair) \frown \mathit{second}(pair)\} \\
\Rightarrow & [3.8] \\
& V/\!\sim = \{pair : rel \bullet (\mathit{first}(pair) \frown \mathit{second}(pair))/\!\sim\}
\end{array}
$$

[1]Note that the symbol $\sim$ is used in two senses in this proof. $\sim$ denotes a data relation, but $rel^{\sim}$ denotes the inverse of the relation rel.

$$
\begin{array}{l}
\Rightarrow \\
V/{\sim} = \{pair : rel \bullet (\mathit{first}(pair)/{\sim}) \frown (\mathit{second}(pair)/{\sim})\} \\
\Rightarrow \\
V/{\sim} = \{pair' : rel' \bullet \mathit{first}(pair') \frown \mathit{second}(pair')\}
\end{array}
$$

To complete the proof that $V/{\sim} \in [\![T \ m\texttt{->}n \ U]\!]_{\delta/\sim}$ it is sufficient to show:

$$
\begin{array}{l}
(\forall\, xs' : [T]_{\delta/\sim} \bullet rel'(\!|\, \{xs'\} \,|\!) \in [\![n \ U]\!]_{\delta/\sim}) \land \\
(\forall\, ys' : [U]_{\delta/\sim} \bullet (rel')^{\sim}(\!|\, \{ys'\} \,|\!) \in [\![m \ T]\!]_{\delta/\sim})
\end{array}
$$

We will prove the first conjunct as the proof of the second is symmetrical. So let $xs' : [T]_{\delta/\sim}$. By A.1 we may choose $xs : [T]_{\delta}$ such that:

$$xs/{\sim} = xs'$$

We will now show:

$$\mathit{Closed}_{\sim}(rel(\!|\, \{xs\} \,|\!))$$

Thanks to A.7 we can use the definition of $\mathit{Closed}_{\sim}$ given in A.6. So let $ys_1 : rel(\!|\, \{xs\} \,|\!)$; $ys_2 : \mathrm{seq}\, Atom$ and suppose:

$$ys_1 \sim ys_2$$

Recall that $\mathit{Closed}_{\sim}(V)$ is a premise of this theorem. Using A.4 we argue:

$$
\begin{array}{ll}
(xs, ys_1) \in rel \land (xs \frown ys_1) \sim (xs \frown ys_2) & \\
\Rightarrow & \\
xs \frown ys_1 \in V \land (xs \frown ys_1) \sim (xs \frown ys_2) & \\
\Rightarrow & [\text{A.6}] \\
xs \frown ys_2 \in V & \\
\Rightarrow & \\
(\exists\, pair : rel \bullet \mathit{first}(pair) \frown \mathit{second}(pair) = xs \frown ys_2 & \\
\quad \land \{xs, \mathit{first}(pair)\} \subseteq [T]_{\delta} & \\
\quad \land \{ys_1, \mathit{second}(pair)\} \subseteq [U]_{\delta} & \\
\quad \land \#(ys_2) = \#(ys_1)) & \\
\Rightarrow & [2.17,\ 2.20] \\
(\exists\, pair : rel \bullet \mathit{first}(pair) \frown \mathit{second}(pair) = xs \frown ys_2 & \\
\quad \land \#(\mathit{first}(pair)) = \#(xs) & \\
\quad \land \#(\mathit{second}(pair)) = \#(ys_2)) & \\
\Rightarrow & \\
(\exists\, pair : rel \bullet \mathit{first}(pair) = xs \land \mathit{second}(pair) = ys_2) &
\end{array}
$$

$\Rightarrow$

$(xs, ys_2) \in rel$

$\Rightarrow$

$ys_2 \in rel(\!|\ \{xs\}\ |\!)$

Completing the proof that:

$$Closed_{\sim}(rel(\!|\ \{xs\}\ |\!))$$

Next we prove:

$$rel'(\!|\ \{xs'\}\ |\!) = (rel(\!|\ \{xs\}\ |\!))/\!\sim$$

So let $ys' : rel'(\!|\ \{xs'\}\ |\!)$. Then:

$$(xs', ys') \in rel'$$

Choose $pair : rel$ such that:

$$xs' = first(pair)/\!\sim \wedge\ ys' = second(pair)/\!\sim$$

It follows:

$first(pair) \frown second(pair) \in V \wedge xs \sim first(pair)$

$\Rightarrow$ [A.4]

$first(pair) \frown second(pair) \in V$

$\quad \wedge\ (first(pair) \frown second(pair)) \sim (xs \frown second(pair))$

$\Rightarrow$ [3.16]

$xs \frown second(pair) \in V$

$\Rightarrow$

$(\exists\, pair_2 : rel \bullet first(pair_2) \frown second(pair_2) = xs \frown second(pair)$

$\quad \wedge\ \{xs, first(pair_2)\} \subseteq [T]_{\delta}$

$\quad \wedge\ \{second(pair), second(pair_2)\} \subseteq [U]_{\delta})$

$\Rightarrow$ [2.17, 2.20]

$(\exists\, pair_2 : rel \bullet first(pair_2) \frown second(pair_2) = xs \frown second(pair)$

$\quad \wedge\ \#(first(pair_2)) = \#(xs)$

$\quad \wedge\ \#(second(pair_2)) = \#(second(pair)))$

$\Rightarrow$

$(\exists\, pair_2 : rel \bullet first(pair_2) = xs \wedge second(pair_2) = second(pair))$

$\Rightarrow$

$(xs, second(pair)) \in rel$

$\Rightarrow$

$$
\begin{array}{l}
second(pair) \in rel(\!|\ \{xs\}\ |\!) \\
\Rightarrow \\
ys' = second(pair)/{\sim} \in (rel(\!|\ \{xs\}\ |\!))/{\sim}
\end{array}
$$

Thus we have shown:

$$rel'(\!|\ \{xs'\}\ |\!) \subseteq (rel(\!|\ \{xs\}\ |\!))/{\sim}$$

Now, to prove the reverse inclusion, let $ys : rel(\!|\ \{xs\}\ |\!)$. Then:

$$
\begin{array}{l}
(xs, ys) \in rel \\
\Rightarrow \\
(xs/{\sim}, ys/{\sim}) \in rel' \\
\Rightarrow \\
(xs', ys/{\sim}) \in rel' \\
\Rightarrow \\
ys/{\sim} \in rel'(\!|\ \{xs'\}\ |\!)
\end{array}
$$

This shows:

$$(rel(\!|\ \{xs\}\ |\!))/{\sim} \subseteq rel'(\!|\ \{xs'\}\ |\!)$$

Hence we have shown:

$$rel'(\!|\ \{xs'\}\ |\!) = (rel(\!|\ \{xs\}\ |\!))/{\sim}$$

Now:

$$
\begin{array}{ll}
rel(\!|\ \{xs\}\ |\!) \in [\![n\ U]\!]_\delta & \\
\Rightarrow & [2.18] \\
rel(\!|\ \{xs\}\ |\!) \in [\![U]\!]_\delta \wedge [\![n]\!](rel(\!|\ \{xs\}\ |\!)) & \\
\Rightarrow & [\text{ind. hyp., A.2}] \\
(rel(\!|\ \{xs\}\ |\!))/{\sim} \in [\![U]\!]_{\delta/\sim} \wedge [\![n]\!]((rel(\!|\ \{xs\}\ |\!))/{\sim}) & \\
\Rightarrow & \\
rel'(\!|\ \{xs'\}\ |\!) \in [\![U]\!]_{\delta/\sim} \wedge [\![n]\!](rel'(\!|\ \{xs'\}\ |\!)) & \\
\Rightarrow & [2.18] \\
rel'(\!|\ \{xs'\}\ |\!) \in [\![n\ U]\!]_{\delta/\sim} &
\end{array}
$$

We have now shown the first conjunct of:

$$
\begin{array}{l}
(\forall\, xs' : [T]_{\delta/\sim} \bullet rel'(\!|\ \{xs'\}\ |\!) \in [\![n\ U]\!]_{\delta/\sim}) \wedge \\
(\forall\, ys' : [U]_{\delta/\sim} \bullet (rel')^{\sim}(\!|\ \{ys'\}\ |\!) \in [\![m\ T]\!]_{\delta/\sim})
\end{array}
$$

and the second is symmetric. This completes the case $T\ m\texttt{->}n\ U$.

□

Lemma A.9
Let $ys : \mathrm{seq}\, Atom$ and $\sim : EqRel$. Then:

$$ys \in (ys/\sim).\sim$$

$\diamond$

Proof The proof uses structural induction.
Case $\langle\rangle$.

$$\begin{array}{ll}
(\langle\rangle/\sim).\sim & \\
= & [3.8] \\
(\langle\rangle).\sim & \\
= & [3.9] \\
\{\langle\rangle\} &
\end{array}$$

Case $\langle y\rangle \frown ys$.

$$\begin{array}{ll}
((\langle y\rangle \frown ys)/\sim).\sim & \\
= & [3.8] \\
(\langle [y]_\sim\rangle \frown (ys/\sim)).\sim & \\
= & [3.9] \\
\{x : [y]_\sim;\ xs : (ys/\sim).\sim \bullet \langle x\rangle \frown xs\} & \\
\ni & \\
\langle y\rangle \frown ys &
\end{array}$$

since $y \in [y]_\sim$ and the inductive hypothesis gives: $ys \in (ys/\sim).\sim$. $\square$

Lemma A.10
Let $\sim : EqRel$ and $xs' : \mathrm{seq}\, Atom^\sim$. Then:

$$(xs'.\sim)/\sim = \{xs'\}$$

$\diamond$

Proof Proof is by structural induction.
Case $\langle\rangle$.

$$\begin{array}{ll}
(\langle\rangle.\sim)/\sim & \\
= & [3.9] \\
\{\langle\rangle\}/\sim & \\
= & [3.8] \\
\{\langle\rangle/\sim\} & \\
= & [3.8]
\end{array}$$

$\{\langle\rangle\}$

Case $\langle x' \rangle \frown xs'$.
Choose $x : x'$. Then $[x]_\sim = x'$. It follows:

$$
\begin{array}{ll}
((\langle x' \rangle \frown xs').\sim)/\sim & \\
= & [3.9] \\
\{y : x';\ ys : (xs'.\sim) \bullet \langle y \rangle \frown ys\}/\sim & \\
= & [3.8] \\
\{y : x';\ ys : (xs'.\sim) \bullet (\langle y \rangle \frown ys)/\sim\} & \\
= & [3.8] \\
\{y : x';\ ys : (xs'.\sim) \bullet (\langle y \rangle/\sim) \frown (ys/\sim)\} & \\
= & [3.8] \\
\{y : x';\ ys : (xs'.\sim) \bullet \langle [y]_\sim \rangle \frown (ys/\sim)\} & \\
= & \\
\{y : [x]_\sim;\ ys : (xs'.\sim) \bullet \langle [y]_\sim \rangle \frown (ys/\sim)\} & \\
= & \\
\{y : [x]_\sim;\ ys : (xs'.\sim) \bullet \langle [x]_\sim \rangle \frown (ys/\sim)\} & \\
= & \\
\{y : x';\ ys : (xs'.\sim) \bullet \langle x' \rangle \frown (ys/\sim)\} & \\
= & \\
\{ys : (xs'.\sim) \bullet \langle x' \rangle \frown (ys/\sim)\} & \\
= & \\
\{zs' : \{ys : (xs'.\sim) \bullet (ys/\sim)\} \bullet \langle x' \rangle \frown zs'\} & \\
= & [3.8] \\
\{zs' : (xs'.\sim)/\sim \bullet \langle x' \rangle \frown zs'\} & \\
= & [\text{ind. hyp.}] \\
\{zs' : \{xs'\} \bullet \langle x' \rangle \frown zs'\} & \\
= & \\
\{\langle x' \rangle \frown xs'\} &
\end{array}
$$

□

Appendix B

Notation

In this appendix we describe the notation used in the thesis which is based on the Z notation [Spi92] [WD96].

B.1 Quantification

We explain how universal and existential quantification is written using examples.

$$\exists\, x : Y \mid Q$$

means there exists x in Y such that Q holds.

$$\forall\, x_1 : Y_1;\ x_2;\ Y_2 \mid P \bullet Q$$

means for all x_1 in Y_1 and x_2 in Y_2 such that P holds, Q holds.

B.2 Set comprehension

A set comprehension is used to define a new sets based on an existing one.

$$\{x : S \mid P\}$$

denotes the set of elements x in the set S such that P holds.

To create a new set whose values are based on some function of the values from an existing set we use the following:

$$\{x : S \mid P \bullet e\}$$

This denotes the set of all expressions e such that x is drawn from S and satisfies P. In most cases e will contain a free occurrence of x.

If P is *true* it can be omitted. That is:

$$\{x : S \mid true \bullet e\} = \{x : S \bullet e\}$$

The declaration part of a set comprehension may introduce more than one variable.

$$\{x : S;\ y : T \mid P \bullet e\}$$

denotes the set of expressions e formed as x and y range over S and T respectively, and satisfy predicate P.

For example

$$\{x : \mathbb{N};\ y : \mathbb{N} \mid x \leq y \land y \leq 2 \bullet x * y\} =$$
$$\{0 * 1, 0 * 2, 1 * 1, 1 * 2\} =$$
$$\{0, 1, 2\}$$

B.3 Symbols

The following table lists some of the symbols used in the Z notation and their meaning.

seq A	the set of sequences of elements from A
$\langle x, y, z \rangle$	the sequence with first element x, second, y and, third, z
$xs \frown ys$	the concatenation of the sequence xs with the sequence ys
$rev(xs)$	the sequence xs, but in the reverse order
$head(xs)$	the first element of the sequence xs
$last(xs)$	the last element of the sequence xs
$A \to B$	the set of functions from A to B
$A \nrightarrow B$	the set of partial functions from A to B
$A \leftrightarrow B$	the set of relations between A and B
$rel^{\sim}$	the inverse (transpose) of the relation rel
$rel(\!\mid X \mid\!)$	the relational image of the set A under the relation R
$\#A$	the number of elements in the set A
$n \mathinner{..} m$	the set of integers between n and m inclusive
$a \mapsto b$	equivalent to (a, b)
$f \oplus (a \mapsto b)$	the function which is like f but maps a to b
$first(pair)$	the first component of the pair $pair$
$second(pair)$	the second component of the pair $pair$

Bibliography

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BS97] Roberto J. Jr. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997.

[DLP79] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, 1979.

[DP02] B. A. Davey and H. A. Priestley. *Introduction to Latices and Order*. Cambridge Univesity Press, 2002.

[EJT04] Jonathan Edwards, Daniel Jackson, and Emina Torlak. A type system for object models. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 189–199, New York, NY, USA, 2004. ACM Press.

[EMGP99] Clarke Edmund M., Jr., Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[FSEL92] Formal Systems (Erope) Ltd. *Failures-Divergences Refinement — FDR2 User Manual*. Formal Systems (Erope) Ltd., 1992.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Ham88] A. G. Hamilton. *Logic for mathematicians*. Cambridge University Press, 1988.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol90] Gerrard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990.

[Jac00] Daniel Jackson. Automating first-order relational logic. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 130–139. ACM Press, 2000.

[Jac02a] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[Jac02b] Daniel Jackson. Micromodels of software: Lightweight modelling and analysis with alloy. Technical report, MIT Lab for Computer Science, February 2002.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2006.

[JJD98] Daniel Jackson, Somesh Jha, and Craig A. Damon. Isomorph-free model enumeration: A new method for checking relational specifications. *ACM Transactions on Programming Languages and Systems*, 20(2):302–343, March 1998.

[Laz99] Ranko S. Lazić. *A Semantic Study of Data Independence with Applications to Model Checking*. PhD thesis, Oxford University Computing Laboratory, 1999.

[LN00] Ranko Lazić and David Nowak. A Unifying Approach to Data-independence. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR 2000)*, volume 1877 of *Lecture Notes in Computer Science*, pages 581–595. Springer-Verlag, August 2000.

[LT93] Nancy Leveson and Clark S. Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[McC01] William McCune. Mace 2.0 reference manual and guide. Technical memorandum, Argonne National Laboratory, May 2001. Mathematics and Computer Science Division, No. 249.

[Mom04] Lee Momtahan. A simple small model theorem for Alloy. Technical report, Oxford University Computing Laboratory, June 2004. PRG-RR-04-11.

[New03] T. C. Newcomb. *Model checking data-independent systems with arrays*. PhD thesis, Oxford University Computing Laboratory, 2003.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[PZ03] Amir Pnueli and Lenore D. Zuck. Model-checking and abstraction to the aid of parameterized systems. In *VMCAI*, page 4, 2003.

[QS83] Jean-Pierre Queille and Joseph Sifakis. Fairness and related properties in transition systems - a temporal logic to deal with fairness. *Acta Inf.*, 19:195–220, 1983.

[RJB99] Jame Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modelling Langauge Reference Manual*. Addison-Wesley, 1999.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[Sla94] J. Slaney. Finder, finite domain enumerator: System description. In *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings*, volume 814 of *Lecture Notes in Computer Science*. Springer, 1994.

[Spi92] J. M. Spivey. *The Z notation : a reference manual*. Prentice-Hall, 1992.

[Spi00] Mike Spivey. The fuzz manual. The Spivey Partnership, 2000. http://spivey.oriel.ox.ac.uk/mike/fuzz/fuzzman.pdf.

[Tur37] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42, pages 230–265, 1937.

[WD96] Jim Woodcock and Jim Davies. *Uzing Z: Specificiation, Refinement, and Proof*. Prentice Hall, 1996.

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

[Wol86] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 184–193. ACM Press, 1986.

[Zha97] Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pages 272–275, 1997.