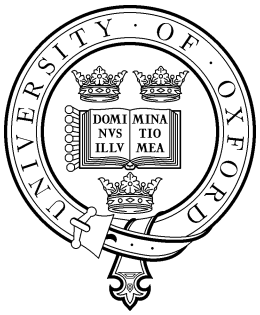
Programming Research Group

A SIMPLE SMALL MODEL THEOREM FOR ALLOY

Lee Momtahan
OXFORD UNIVERSITY COMPUTING LABORATORY

PRG-RR-04-11



Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

Alloy is an extension of first-order logic for modelling software systems. Alloy has a fully automatic analyser which attempts to refute Alloy formulae by searching for counterexamples within a finite scope. However, failure to find a counterexample does not prove the formula correct. 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 paper gives a theorem in a language closely related to Alloy, which applies to models of data-independent systems. The theorem calculates for such types T a threshold size. If no counterexamples are found at the threshold, the theorem guarantees that increasing the scope on T beyond the threshold still yields no counterexamples, and can complete the analysis for data-independent systems.

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 prone to failure. Failure of computer systems can cause a significant economic impact and even loss of life.

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 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 naive 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*.

In particular this transfer thesis [1] looks at a technique known as *model finding*. Model finders are programs which attempt to refute a logic formula by searching for a model of the formula's negation. Such a model corresponds to a failed test case of the system in question, and model finding can be thought of as exhaustive, automated testing. Being completely automatic, model finders can reduce the labour costs of the formal methods approach significantly. However, to test exhaustively the system in question in a finite time, one has to supply the model finder with a *scope*, which tells the model finder the size of base types to use in its search. This has the effect of approximating infinite systems by finite ones. If no counterexamples are found in the finite approximation, this gives an indication that the infinite system and larger finite systems are correct, but no guarantee. Model finders can not guarantee correctness of infinite systems.

The problem to be addressed in the thesis proposed herein is the extension of model finding technique for guaranteeing correctness from finite systems, to systems where infinite data independent types are also allowed. Informally and operationally speaking, a system is data independent in a type T if it can only input, output, store and perform equality tests on the values of T. No other operations are permitted. We will look at this problem in the particular context of the Z-like language Alloy [Jac02a] and its associated model finder.

[1] This technical report was originally the author's transfer thesis under the title: Towards a Small Model Theorem for Data Independent Systems in Alloy.

1.1 Overview

In section 2 we look in more depth at related work in computer-aided verification. In the following section, we look at a motivating example and show informally how the small model theorem would work here. In section 4 we prove a small model theorem on a language related to Alloy. Although this is a significant step towards solving the proposed thesis problem, there are minor differences between this language and the language required to solve the thesis problem. Therefore we conclude by outlining the further work required to solve the problem, and other potential directions this research could then take.

2 Related Work

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

Recall that 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*).

2.1 Theorem Proving

With theorem proving the specification and implementation are represented in formalisms which allow the refinement of the specification by the implementation to be represented as a logic formula; then axioms and inference rules are used to prove the formula. For 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, although it can be shown impossible to completely automate the process — this follows trivially from the *Halting Problem*.

2.1.1 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 *parameterised* systems. [See Sec. 2.4]. The main disadvantage with theorem proving is that a high degree of highly skilled labour is required, even when proof tools are used.

2.2 Model Checking

This is a brute-force approach to verifying correctness, where a complete state-transition graph of the implementation is constructed and automatically checked state by state that is satisfies its specification. SPIN [Hol90] and FDR [FSEL92] are model checkers, for example.

2.2.1 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, one can construct a complete state-transition graph only when the system is *finite state*. Furthermore, some systems suffer from *state-explosion*, which means 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.

2.3 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.

2.3.1 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.

2.4 Infinite and Parameterised 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 parametrised systems where the value of a parameter ranges over an infinite range. 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 parameterised systems are ones which require checks where it not possible to give a priori bounds on the size of types. This is clearly an analogous 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 follows immediately from the *Halting Problem* that is it impossible to build a model checker or finder which can in general deal with infinite-state systems or the general parameterised verification problem. Nevertheless an active research area is finding special

cases of these problems which can be dealt with. These techniques can not only reduce the state space from infinite to finite, but for some problems reduce state-explosion.

2.4.1 Data Independence

One special case 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, and uninterpreted constants and finite range functions on the type as well, whilst 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.

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 up to a particular value, then the system is correct for all non-empty instantiation of the type [Wol86].

2.5 Alloy and the Alloy Analyser

The Alloy [Jac02a] [Jac02b] modelling 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 [JRB99][WK99]. Alloy is designed to bring to Z-style specification the kind of automation offered by model checkers.

The Alloy Analyser is a model finder for the Alloy language. The Analyser works by first translating formulae in the Alloy Language into a smaller Alloy Kernel language [Jac02a]. Because the Analyser is given a finite *scope* (Sec. 2.3.1), 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 Analyser wraps off-the-shelf SAT solvers, such as SATO [Zha97] or RelSAT [BS97].

2.6 MACE and other Model Finders

Other model finders such as MACE [McC] 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. Unfortunately, the theory we will develop does not apply to such languages except for a trivial class of problems.

2.7 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 below a particular threshold size. The notion of size in our case is scope, which we define formally in Sec. 4.29. 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 proceeding theory is inspired by this intuition, and aspects of it have similarities with data independence theory in model checking, in particular [LN00]. Although our approach is model theoretic, we are not aware of known results from model theory which could be directly applied.

3 Example and Motivation

In this section we give an example problem and *informally* develop a small model theorem for this particular example. The approach used can be applied to a general class of problems though, and so we formally develop a general small model theorem in the following section. The example is intended to illustrate the thesis problem proposed in Sec. 1, and provide intuition for the more general theorem.

3.1 Birthday Book

The example is the Birthday Book from [Spi92]:

$[NAME, DATE]$

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$
$$known = \mathrm{dom}(birthday)$$

AddBirthday

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$
$$birthday' = birthday \oplus \{name? \mapsto date?\}$$

FindBirthday

$$\Xi BirthdayBook$$
$$name? : NAME$$
$$date! : DATE$$
$$date! = birthday(name?)$$

DeleteBirthday

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$birthday' = \{name?\} \rhd birthday$$

Let us suppose we want to check the following assertions which we will refer to as *AddWorks* and *DelIsUndo* respectively:

$$AddBirthday \fatsemi FindBirthday \Rightarrow date? = date!$$

$$AddBirthday \fatsemi DeleteBirthday \Rightarrow \Xi BirthdayBook$$

Jackson translates this into the Alloy language as follows:

```
module samples/BirthdayBook

sig Name {}
sig Date {}
sig BirthdayBook {known: set Name, birthday: known ->! Date}

fun AddBirthday (bb, bb': BirthdayBook, n: Name, d: Date) {
        bb'.birthday = bb.birthday ++ (n->d)
        }

fun DelBirthday (bb, bb': BirthdayBook, n: Name) {
        bb'.birthday = bb.birthday - (n->Date)
        }

fun FindBirthday (bb: BirthdayBook, n: Name, d: option Date) {
        d = bb.birthday[n]
        }

assert AddWorks {
        all bb, bb': BirthdayBook, n: Name, d: Date, d': option Date |
                AddBirthday (bb,bb',n,d) && FindBirthday (bb',n,d') => d = d'
         }


assert DelIsUndo {
        all bb1,bb2,bb3: BirthdayBook, n: Name, d: Date|
                AddBirthday (bb1,bb2,n,d) && DelBirthday (bb2,bb3,n)
                        => bb1.birthday = bb3.birthday && bb1.known=bb2.known
        }

check AddWorks for 3 but 2 BirthdayBook
check DelIsUndo for 3 but 2 BirthdayBook
```

The statement: *check DelIsUndo for 3 but 2 BirthdayBook* instructs the Alloy Analyser to look for a counterexample to the *DelIsUndo* assertion in which the carrier set size of *Name* and *Date* are 3 but *BirthdayBook* is 2. This produces the following counterexample:

```
BirthdayBook = {BirthdayBook_0, BirthdayBook_1}
Date = {Date_0, Date_1, Date_2}
Name = {Name_0, Name_1, Name_2}
```

```
known = {BirthdayBook_0 -> {},
         BirthdayBook_1 -> {Name_2} }
birthday = {BirthdayBook_0 -> {},
            BirthdayBook_1 -> {Name_2 -> Date_2} }

bb1@1 = BirthdayBook_1
bb2@2 = BirthdayBook_1
bb3@3 = BirthdayBook_0
n@4 = Name_2
d@5 = Date_2
```

We know the *DelIsUndo* assertion is false.

The statement *check AddWorks for 3 but 2 BirthdayBook* similarly instructs the Alloy Analyser to look for a counterexample to the *AddWorks* assertion. None is found so we don't know whether the assertion is true or false. Could we find a counterexample if we increase the scope? At first we ask the following question: Keeping the size of *BirthdayBook* at 2 and *Name* at 3 could we find a counterexample to the *AddWorks* assertion by increasing the size of *Date*, and by how much?

Returning to the the *DelIsUndo* counterexample for a moment, we see the variables *bb1@1, bb2@2, bb3@3, n@4, d@5* in the counterexample. These are called skolem variables and the first thing the Alloy Analyser does with a formula is convert to negation normal form and skolemize [Ham88] [Jac00].

Our argument to generate a threshold on *Date* in the *AddWorks* check proceeds by a similar process of Skolemization. Let us suppose a counterexample (countermodel) does exist. We can fill in the following table with 0 and 1's to represent the assignment of variables involving *Date* in the counterexample. There is one row in the table per value of *Date*, so potentially an infinite number of rows in the table. The skolem variables are *d@1* and *d'@2*.

	BirthdayBook_0			*BirthdayBook_1*				
	Name_0	*Name_1*	*Name_2*	*Name_0*	*Name_1*	*Name_2*	*d@1*	*d'@2*
Date_0								
Date_1								
...								
Date_n								
...								

Let us consider a typical assignment.

	BirthdayBook_0			*BirthdayBook_1*				
	Name_0	*Name_1*	*Name_2*	*Name_0*	*Name_1*	*Name_2*	*d@1*	*d'@2*
Date_0	1	0	0	0	0	0	0	0
Date_1	0	1	0	0	0	0	0	0
Date_2	0	0	1	0	0	0	0	0
Date_3	0	0	0	1	0	0	0	0
Date_4	0	0	0	0	1	0	0	0
Date_5	0	0	0	0	0	1	0	0
Date_6	0	0	0	0	0	0	1	0
Date_7	0	0	0	0	0	0	0	1
Date_8	0	0	0	0	0	0	0	0
Date_9	0	0	0	0	0	0	0	0
...	0	0	0	0	0	0	0	0
Date_∞	0	0	0	0	0	0	0	0

We can transform this assignment as follows:

	BirthdayBook_0			*BirthdayBook_1*				
	Name_0	*Name_1*	*Name_2*	*Name_0*	*Name_1*	*Name_2*	*d@0*	*d'@1*
{Date_0}	1	0	0	0	0	0	0	0
{Date_1}	0	1	0	0	0	0	0	0
{Date_2}	0	0	1	0	0	0	0	0
{Date_3}	0	0	0	1	0	0	0	0
{Date_4}	0	0	0	0	1	0	0	0
{Date_5}	0	0	0	0	0	1	0	0
{Date_6}	0	0	0	0	0	0	1	0
{Date_7}	0	0	0	0	0	0	0	1
{Date_8... Date_∞}	0	0	0	0	0	0	0	0

The resulting table is formed by choosing values of *Date* which are the equivalence classes of the values of *Date* having equivalent rows in the table. These actual values of *Date*, referred to as *atoms*, are not important to the countermodel; all that matters of course is that there are nine of them and they are distinct. The above transformation can be repeated on any potential assignment. In the former table, '1' may occur no more than once in each column, so there are no more than 8 rows, which do not contain '0' all the way through. This tells us there will never be more than 9 equivalence classes for any assignment, so a maximum of 9 rows in the latter table.

Because the formula we are checking does not use quantifier or set builder constructs over the *Date* type, it turns out that the original assignment is a countermodel if and only if the transformed assignment is a countermodel (with the assignment of variables not covered by the table i.e. not involving *Date*, unchanged). This tells us that 9 is a threshold for *Date*.

If the scope on *BirthdayBook* or *Name* were to increase, we would have to increase the threshold on *Date* accordingly, and we could easily give the appropriate function. We have not generated an overall threshold scope, but this may be possible. [See Sec.5.2].

We would like to stress our argument applies to a general class of formulae. We can draw a similar table for any formula and if there are a finite number of columns, there will be a finite number of equivalence classes. This is so, provided we assume a finite scope on all type variables other than *Date*, and each variable in the formula is declared with a type expression which uses *Date* no more than once. The other condition is that the formula does not use set builder or quantifier constructs over *Date*; then the number of equivalence classes is a threshold. This latter condition can be weakened using Skolemization. We can of course repeat the argument for type variables other than *Date*.

4 Small Model Theorem

In this section we prove our small model theorem which will be described formally in 4.32. Without loss of generality, we will assume the type variable for which we want to obtain a bound is X. The small model theorem takes a formula, a declaration of the type of each variable, and a scope for each relevant type variable other than X. Subject to various side conditions, the theorem gives a scope for X such that a check at this *threshold* scope for X is equivalent to one with a larger scope for X.

Since the full Alloy language is compiled by the Analyser into a much smaller Alloy Kernel language, we choose to make a small model theorem about this language. We considered a developing a small model theorem for the full Alloy language or Z, but these languages have a much larger syntax and this would have added significant burden to our proof. For now, it seems progress can be made more quickly with a smaller language; other languages could be considered later. In fact the language we give here has some differences from the Alloy Kernel language. We present these differences and discuss how they could be eliminated in 5.1. Since we present this language in a rather terse manner, the reader may wish to refer to [Jac02a].

The notation we use for the meta-language in this section is roughly based on Z [Spi92].

4.1 Definition: Type Syntax

Let $\mathit{TypeVar}$ be a set, whose elements are called type variables. Let $M == \{*, !, ?\}$, the set of multiplicity markings. A type expression takes the following form:

$$\mathit{TypeExp} ::= \mathit{TypeVar}\ M \rightarrow M\ \mathit{TypeVar}$$

In other words a type expression is formed as a quadruple of values
$A : \mathit{TypeVar},\ B : \mathit{TypeVar},\ m : M,\ n : M$ and written $A\ m \rightarrow n\ B$. In this context $\rightarrow$ is an uninterpreted symbol of the meta-language.

4.2 Definition: Free Types

The following function returns the type variables used in a a type expression:

$$\mathit{Free}(A\ m \rightarrow n\ B) = \{A, B\}$$

4.3 Definition: Set Map

Let Atom be a set, whose elements we will call atoms. A set map is a partial map from type variables to sets of atoms. Thus the set of set maps is:

$$\mathit{SetMap} == \mathit{TypeVar} \nrightarrow \mathbb{P}\,\mathit{Atom}$$

The image of a type variable under this map is called its carrier set.

4.4 Definition: Type Semantics

Given $T \in \mathit{TypeExp}$ and $\delta \in \mathit{SetMap}$ such that $\mathit{Free}(T) \subseteq \operatorname{dom}\delta$, the denotational semantics of T with respect to δ, written $[\![T]\!]_\delta$ is defined as follows[2]:

$$
\begin{array}{l}
[\![A\ m \rightarrow n\ B]\!]_\delta == \\
\quad \{r : \mathbb{P}(\delta(A) \times \delta(B)) \mid \\
\qquad\qquad (m = ! \Rightarrow \forall\, b : \delta(B) \bullet \#\operatorname{dom}(r \rhd \{b\}) = 1) \\
\qquad\qquad \land\ (m = ? \Rightarrow \forall\, b : \delta(B) \bullet \#\operatorname{dom}(r \rhd \{b\}) \leq 1) \\
\qquad\qquad \land\ (n = ! \Rightarrow \forall\, a : \delta(A) \bullet \#\operatorname{ran}(\{a\} \lhd r) = 1) \\
\qquad\qquad \land\ (n = ? \Rightarrow \forall\, a : \delta(A) \bullet \#\operatorname{ran}(\{a\} \lhd r) \leq 1)) \\
\qquad\qquad \}
\end{array}
$$

The type semantics gives the meaning of a type expression. We use the type semantics to tell us what possible meanings a variable of the type expression can have.

4.5 Example: Type Semantics

Note, the multiplicity markings can be used to refer to general relationships between types, or more specific relationships, such as partial/total, injective/surjective functions e.g.

$$
\begin{array}{l}
[\![A * \rightarrow * B]\!]_\delta = \delta(A) \leftrightarrow \delta(B) \\
[\![A * \rightarrow ?\ B]\!]_\delta = \delta(A) \nrightarrow \delta(B) \\
[\![A\ ! \rightarrow !\ B]\!]_\delta = \delta(A) \rightarrowtail \delta(B)
\end{array}
$$

4.6 Definition: Type Map

Let Var be a set whose elements we will call variables. A type map is a partial map from variables to type expressions whose domain is finite[3]:

$$\mathit{TypeMap} == \{\Gamma : \mathit{Var} \nrightarrow \mathit{TypeExp} \mid \operatorname{dom}\Gamma \in \mathbb{F}\ \mathit{Var}\}$$

The type map is used to determine the types of variables.

[2] the symbols $\rhd, \lhd$ are respectively the range and domain restriction, see [Spi92]

[3] $\mathbb{F}$ denotes the set of finite subsets of some set

4.7 Definition: Signature

A signature is an ordered pair, (Υ, Γ), where:

$$\Upsilon \subseteq \mathit{TypeVar}$$
$$\Gamma \in \mathit{TypeMap}$$
$$\forall\, v : \operatorname{dom}\Gamma \bullet \mathit{Free}(\Gamma(v)) \subseteq \Upsilon$$

A formula or expression gives rise to a compatible signature which will be defined later in 4.24. What the signature tells us, for the formula or expression at hand is: the relevant type variables (Υ), the relevant variables ($\operatorname{dom}\Gamma$), and the types of those variables (Γ). The final condition ensures that the set of relevant type variables includes those appearing in the type expression of any relevant variable.

4.8 Example: Signature

An example signature (Υ, Γ) is:

$$\Upsilon = \{U, X\}$$
$$\Gamma = \{v \mapsto (U * \rightarrow! X), w \mapsto (U * \rightarrow * X)\}$$

In this case the signature gives the following information. The type of the variable v is a total function from U to X. The type of the variable w is a relation between U and X. The relevant type variables are U and X.

4.9 Definition: Instantiation

Given a signature $S = (\Upsilon, \Gamma)$, an instantiation of S is an ordered pair, (δ, η), where:

$$\delta \in \mathit{SetMap}$$
$$\eta \in \mathit{Var} \nrightarrow \mathbb{P}(\mathit{Atom} \times \mathit{Atom})$$
$$\operatorname{dom}\delta = \Upsilon$$
$$\operatorname{dom}\eta = \operatorname{dom}\Gamma$$
$$\forall\, v : \operatorname{dom}\eta \bullet \eta(v) \in [\![\Gamma(v)]\!]_\delta$$

The instantiation is used to give a denotational semantics to the expressions and formulae of the language. The instantiation tells us the carrier sets of the relevant type variables. It also tells us the meaning of any variables. The last condition ensures that the meaning of each variable is consistent with the semantics of its type.

4.10 Example: Instantiation

(δ, η) is an instantiation of the above signature (4.8), where:

$$\delta = \{U \mapsto \{a\}, X \mapsto \{1,2,3,4,5\}\}$$
$$\eta = \{v \mapsto \{(a,1)\}, w \mapsto \{(a,1),(a,2)(a,3)\}\}$$

The carrier set of U is {a}, and X is {1,2,3,4,5}. The meaning given to v by η is indeed a total function form $\delta(U)$ to $\delta(X)$. The meaning given to w is a relation between $\delta(U)$ and $\delta(X)$. Thus this is a valid instantiation of the signature.

4.11 Definition: Distinguished Type Variable

X is a fixed element of *TypeVar*. X will be the candidate type variable on which we will apply our threshold theorem. We will be developing the appropriate conditions for the theorem to be applicable as we proceed.

4.12 Definition: Related Atoms

Recall X is a fixed element of *TypeVar*. Given a signature, $S = (\Upsilon, \Gamma)$, and a instantiation of it, $I = (\delta, \eta)$, for each type variable $Y \in \Upsilon$, we define a relation on $\delta(Y)$, called $\sim^Y_{S,I}$ as follows:

$$y \sim^Y_{S,I} y' \Leftrightarrow y = y' \qquad (\text{provided } Y \neq X)$$

$$x \sim^X_{S,I} x' \Leftrightarrow \forall(v, Y\ m \to n\ Z) : \Gamma \bullet$$
$$(Y = X \Rightarrow \text{ran}(\{x\} \lhd \eta(v)) = \text{ran}(\{x'\} \lhd \eta(v))) \wedge$$
$$(Z = X \Rightarrow \text{dom}(\eta(v) \rhd \{x\}) = \text{dom}(\eta(v) \rhd \{x'\}))$$

When $x \sim^X_{S,I} x'$, no variable v can witness any difference between x and x'. Although perhaps not equal, x and x' are completely interchangeable as elements of the carrier set; neither atom has any property the other does not.

For further intuition the reader is referred back to the tabular presentation in the example (sec. 3.1) and the notion of equivalent rows.

4.13 Lemma: Related Atoms are Equivalent

Claim: For any signature $S = (\Upsilon, \Gamma)$, instantiation of it, $I = (\delta, \eta)$, and type variable $Y \in \Upsilon$, $\sim^Y_{S,I}$ is an equivalence relation.

Proof: Trivial.

4.14 Convention: Equivalence classes

Given an equivalence relation $\sim$ on a set S, for any $s \in S$ let $[s]_\sim$ be the equivalence class of s, i.e.

$$[s]_\sim == \{s' : S \mid s \sim s'\}$$

Let $[]_\sim$ be the set of all equivalence classes, i.e.

$$[]_\sim == \{s : S \bullet [s]_\sim\}$$

4.15 Example: Atom Equivalence Classes

If we use the same example signature and instantiation (4.8, 4.10):

$$\Upsilon = \{U, X\}$$
$$\Gamma = \{v \mapsto (U * \rightarrow! X), w \mapsto (U * \rightarrow * X)\}$$
$$\delta = \{U \mapsto \{a\}, X \mapsto \{1, 2, 3, 4, 5\}$$
$$\eta = \{v \mapsto \{(a, 1)\}, w \mapsto \{(a, 1), (a, 2)(a, 3)\}$$

the atom equivalence classes in the carrier set of X will be:

$$[]_{\sim^X_{(\Upsilon,\Gamma),(\delta,\eta)}} = \{\{1\}, \{2, 3\}, \{4, 5\}\}$$

$\{1\}$ forms an equivalence class because (a, x) is in $\eta(v)$ only when $x = 1$. $\{2, 3\}$ forms an equivalence class because (a, x) is in $\eta(w)$ and not in $\eta(v)$ only when $x = 2 \vee x = 3$. $\{4, 5\}$ forms an equivalence class because (a, x) is not in $\eta(w)$ and not in $\eta(v)$ only when $x = 4 \vee x = 5$.

4.16 Definition: Size

Given a signature $S = (\Upsilon, \Gamma)$, and instantiation of it, $I = (\delta, \eta)$, we define the size, $Size(S, I)$:

$$Size(S, I) == \infty \qquad \text{if } \exists(v, A\ m \rightarrow n\ B) : \Gamma \bullet A = X \wedge B = X$$
$$Size(S, I) == \textstyle\sum_{(v,T):\Gamma} Sum(T) + \prod_{(v,T):\Gamma} Prod(T) \qquad \text{otherwise}$$

where, we define $Sum(T)$ for any type expression T to be 0, except for any $A : TypeVar \setminus \{X\}, m : M$:

$$Sum(A\ m \rightarrow! X) == \#\delta(A)$$
$$Sum(X\ ! \rightarrow m\ A) == \#\delta(A)$$
$$Sum(A\ m \rightarrow? X) == \#\delta(A)$$
$$Sum(X\ ? \rightarrow m\ A) == \#\delta(A)$$

and we define $Prod(T)$ for any type expression T to be 1, except for any $A : TypeVar \setminus \{X\}$:

$$Prod(A\ !\rightarrow *\ X) == \#\delta(A)$$
$$Prod(X\ *\rightarrow !\ A) == \#\delta(A)$$
$$Prod(A\ ?\rightarrow *\ X) == \#\delta(A) + 1$$
$$Prod(X\ *\rightarrow ?\ A) == \#\delta(A) + 1$$
$$Prod(A\ *\rightarrow *\ X) == 2^{\#\delta(A)}$$
$$Prod(X\ *\rightarrow *\ A) == 2^{\#\delta(A)}$$

Note that $Size(S, I)$ is dependent only on S and $\#\delta(Y)$ for $Y \in \Upsilon\setminus\{X\}$. It is independent of η and $\delta(X)$.

4.17 Theorem: Number of Atom Equivalence Classes

Claim: For any signature, S, and instantiation of it, I, we have:

$$\#[]_{\sim^X_{S,I}} \leq Size(S, I)$$

Proof: Let $S = (\Upsilon, \Gamma)$ be a signature and $I = (\delta, \eta)$ be an instantiation of it. We may assume there is no element of the form $X\ n \rightarrow m\ X$ in $\operatorname{ran}\Gamma$ for then $Size(S, I) = \infty$ and there is nothing to prove.

For each $v : Var;\ A : TypeVar;\ m : M;\ n : \{!, ?\}$ such that $(v, A\ m \rightarrow n\ X) \in \Gamma$ and each $a : \delta(A)$ we define the following predicates on $\delta(X)$:

$$P_{(v,a)}(x) \Leftrightarrow (a, x) \in \eta(v)$$

Similarly for each $v : Var;\ A : TypeVar;\ m : M;\ n : \{!, ?\}$ such that $(v, X\ n \rightarrow m\ A) \in \Gamma$ and each $a : \delta(A)$ we define the following predicates on $\delta(X)$:

$$P_{(v,a)}(x) \Leftrightarrow (x, a) \in \eta(v)$$

Thus we have defined $\sum_{(v,T):\Gamma} Sum(T)$ (not necessarily distinct) predicates. Each predicate determines a set: the set of atoms in $\delta(X)$ for which it holds. Due to the restriction on n, the multiplicity marking associated with X, we see each set is at most a singleton. If the set is a singleton, $\{x\}$ say, and $x' \notin \{x\}$ then $\neg\ P_{(v,a)}(x')$ so $\neg\ x' \sim^X_{S,I} x$ by 4.12. Thus each set is either empty or an equivalence class.

Let Q be the disjunction of the above predicates. Consider the functions $f : Var \rightarrow \mathbb{P}\, Atom$ with the following constraints:

$$\begin{aligned}&\operatorname{dom} f = \{v : Var \mid \\ &\ \exists\, m : M;\ \ A : TypeVar \mid (v, A\ m \rightarrow *\ X) \in \Gamma \vee (v, X\ * \rightarrow m\ A) \in \Gamma \\ &\}\end{aligned}$$

(i.e. the domain of the function is the set of variables mapped by Γ to types containing $* X$)

$$
\begin{array}{l}
\forall (v, as) : f \bullet \\
\quad \exists\, m : M;\ \ A : \mathit{TypeVar} \mid (v, A\ m \to * X) \in \Gamma \lor (v, X * \to m\ A) \in \Gamma \bullet \\
\qquad (m =! \Rightarrow as \in \{a : \delta(A) \bullet \{a\}\}) \land \\
\qquad (m =? \Rightarrow as \in \{a : \delta(A) \bullet \{a\}\} \cup \{\emptyset\}) \land \\
\qquad (m = * \Rightarrow as \in \mathbb{P}\, A)
\end{array}
$$

There are $\prod_{(v,T):\Gamma} \mathit{Prod}(T)$ such functions. We let each one define a predicate on $\delta(X)$ as follows:

$$
\begin{array}{l}
P_f(x) \Leftrightarrow \\
\quad (\forall\, v : \mathit{Var} \mid \ (\exists\, A : \mathit{TypeVar};\ m : M \bullet (v, A\ m \to * X) \in \Gamma) \bullet \\
\qquad\qquad\qquad \mathrm{dom}(\eta(v) \rhd \{x\}) = f(v)) \\
\qquad\qquad \land \\
\quad (\forall\, v : \mathit{Var} \mid \ (\exists\, A : \mathit{TypeVar};\ m : M \bullet (v, * X \to m\ A) \in \Gamma) \bullet \\
\qquad\qquad\qquad \mathrm{ran}(\{x\} \lhd \eta(v)) = f(v)) \\
\qquad\qquad \land \\
\qquad\quad \neg\ Q
\end{array}
$$

Again, each predicate determines a set. If this set is non-empty, with reference to 4.12 we see it is an equivalence class.

Furthermore, if $x \in \delta(X)$ and not $Q(x)$ then define $g : \mathit{Var} \to \mathbb{P}\, \mathit{Atom}$ by:

$$
\begin{array}{l}
\{v : \mathit{Var};\ A : \mathit{TypeVar};\ m : M \mid \\
\qquad\qquad (v, A\ m \to * X) \in \Gamma \bullet (v, \mathrm{dom}(\eta(v) \rhd \{x\}))\} \\
\cup \\
\{v : \mathit{Var};\ A : \mathit{TypeVar};\ m : M \mid \\
\qquad\qquad (v, X * \to m\ A) \in \Gamma \bullet (v, \mathrm{ran}(\{x\} \lhd \eta(v)))\}
\end{array}
$$

which satisfies the above constraints on functions from Var to $\mathbb{P}\, \mathit{Atom}$ due to the multiplicity markings. Clearly P_g holds for x. Thus we have shown that the totality of predicates (of the form $P_{(v,a)}$ and P_f) given above are exhaustive.

It follows that every equivalence class of $\sim^X_{S,I}$ is determined by at least one of the above predicates, of which there are $\mathit{Size}(S, I)$. This completes the proof.

4.18 Example: Applying the above Theorem

With the assignments of 4.15 we have $\mathit{Size}((\Upsilon, \Gamma), (\eta, \delta)) = \#\delta(U) + 2^{\#\delta(U)} = 3$. The above theorem tells us there are no more than 3 equivalence classes in $[]_{\sim^X_{S,I}}$. Furthermore because $\mathit{Size}((\Upsilon, \Gamma), (\eta, \delta))$ is independent of η and $\delta(X)$, changing their values will never give more than 3 equivalence classes.

4.19 Definition: Quotient Instantiation

In the context of a given a signature $S = (\Upsilon, \Gamma)$, we define on any instantiation of it $I = (\delta, \eta)$, the quotient instantiation $\overline{I} = (\overline{\delta}, \overline{\eta})$ as follows:

$$\begin{array}{l}
\operatorname{dom} \overline{\delta} == \Upsilon \\
\forall\, A : \Upsilon \bullet \overline{\delta}(A) == []_{\sim^A_{S,I}} \\
\operatorname{dom} \overline{\eta} == \operatorname{dom} \Gamma \\
\forall (v, A\ m \to n\ B) : \Gamma \bullet \overline{\eta}(v) == \{(a, b) : \eta(v) \bullet ([a]_{\sim^A_{S,I}}, [b]_{\sim^B_{S,I}})\}
\end{array}$$

The quotient instantiation uses an alternate set of atoms, $\overline{Atom}$, where:

$$\overline{Atom} = \mathbb{P}\, Atom$$

In the context of a quotient instantiation, we replace $Atom$ with $\overline{Atom}$ in the definition of set map (4.3) and instantiation (4.9). For the sake of brevity we do not repeat these definitions.

The quotient instantiation will be useful for us because there are two ways to see the atoms it uses. As atoms of the quotient instantiation there is a completely flat structure on them; they can only be compared with each other for equality and no more. But as sets of atoms of the original instantiation they will allow us to make a link between the original and quotient instantiations.

4.20 Lemma: Quotient Instantiation is a Valid Instantiation

Claim: If S is a signature and I and instantiation of S, then $\overline{I}$ is also an instantiation of S.

Proof: Let $S = (\Upsilon, \Gamma)$ be a signature and $I = (\delta, \eta)$ an instantiation of it. The conditions for $\overline{I}$ to be a valid instantiation, given in 4.9, follow trivially from 4.19 with the exception of $\forall\, v : \operatorname{dom} \overline{\eta} \bullet \overline{\eta}(v) \in [\![\Gamma(v)]\!]_{\overline{\delta}}$, which we now prove.

Recall the definition of type semantics (4.4):

$$\begin{array}{l}
[\![A\ m \to n\ B]\!]_\delta == \\
\quad \{r : \mathbb{P}(\delta(A) \times \delta(B)) \mid \quad (m = ! \Rightarrow \forall\, b : \delta(B) \bullet \#\operatorname{dom}(r \rhd \{b\}) = 1) \\
\qquad\qquad\qquad\qquad \land \quad (m = ? \Rightarrow \forall\, b : \delta(B) \bullet \#\operatorname{dom}(r \rhd \{b\}) \leq 1) \\
\qquad\qquad\qquad\qquad \land \quad (n = ! \Rightarrow \forall\, a : \delta(A) \bullet \#\operatorname{ran}(\{a\} \lhd r) = 1) \\
\qquad\qquad\qquad\qquad \land \quad (n = ? \Rightarrow \forall\, a : \delta(A) \bullet \#\operatorname{ran}(\{a\} \lhd r) \leq 1)) \\
\qquad\qquad\qquad\qquad \}
\end{array}$$

Let $(v, A\ m \to n\ B) : \Gamma$.

$$\begin{aligned}\overline{\eta}(v) &= \{(a,b) : \eta(v) \bullet ([a]_{\sim^A_{S,I}}, [b]_{\sim^B_{S,I}})\} \qquad (4.19)\\ &\subseteq \{(a,b) : \delta(A) \times \delta(B) \bullet ([a]_{\sim^A_{S,I}}, [b]_{\sim^B_{S,I}})\} \quad (4.9)\\ &= \overline{\delta}(A) \times \overline{\delta}(B)\end{aligned}$$

Suppose for some $b' : \overline{\delta}(B)$ we have $\#\operatorname{dom}(\overline{\eta}(v) \rhd \{b'\}) > 1$. Choose $a'_1, a'_2 : \overline{\delta}(A)$ such that $a'_1 \neq a'_2$ and $(a'_1, b') \in \overline{\eta}(v)$ and $(a'_2, b') \in \overline{\eta}(v)$. Then choose $(a_1, b_1), (a_2, b_2) : \eta(v)$ such that $([a_1]_{\sim^A_{S,I}}, [b_1]_{\sim^B_{S,I}}) = (a'_1, b')$ and $([a_2]_{\sim^A_{S,I}}, [b_2]_{\sim^B_{S,I}}) = (a'_2, b')$. It follows that $b_1 \sim^B_{S,I} b_2$, and so (by 4.12) $(a_1, b_2) \in \eta(v)$. Since a_1 and a_2 are in distinct equivalence classes, $a_1 \neq a_2$, and therefore $\#\operatorname{dom}(\eta(v) \rhd \{b_2\}) > 1$. This implies $m \neq!$ and $m \neq?$. We have shown the following:

$$m =! \vee m =? \Rightarrow \forall\, b' : \overline{\delta}(B) \bullet \#\operatorname{dom}(\overline{\eta}(v) \rhd \{b\}) \leq 1 \qquad (1)$$

Now suppose $m =!$. For all $b : \delta(B)$ we have $\#\operatorname{dom}(r \rhd \{b\}) > 0$. Let $b' : \overline{\delta}(B)$. Choose $b : b'$. Since $b \in \delta(B)$ we can choose $a : \delta(A)$ such that $(a, b) \in \eta(v)$. Then let $a' = [a]_{\sim^A_{S,I}}$. It now follows $(a', b') \in \overline{\eta}(v)$ and thus for all $b' : \overline{\delta}(B)$, $\#\operatorname{dom}(r \rhd \{b\}) > 0$. Therefore we have shown:

$$m =! \Rightarrow \forall\, b' : \overline{\delta}(B) \bullet \#\operatorname{dom}(\overline{\eta}(v) \rhd \{b'\}) > 0 \qquad (2)$$

Combining (1) and (2) we have:

$$\begin{aligned}&m =! \Rightarrow \forall\, b' : \overline{\delta}(B) \bullet \#\operatorname{dom}(\overline{\eta}(v) \rhd \{b'\}) = 1\\ &m =? \Rightarrow \forall\, b' : \overline{\delta}(B) \bullet \#\operatorname{dom}(\overline{\eta}(v) \rhd \{b'\}) \leq 1\end{aligned}$$

And by appealing to symmetry, we see $\overline{\eta}(v) \in [\![\Gamma(v)]\!]^{\overline{\eta}}_{\overline{\delta}}$ according to the definition of type semantics (4.4) we recalled above. Since $\operatorname{dom}\overline{\eta} = \operatorname{dom}\Gamma$, we have now completed the proof.

4.21 Definition: Unit Type Variable

There is a special element $\mathit{Unit} \in \mathit{TypeVar}$, which always has a singleton carrier set i.e.:

$$\forall\, \delta : \mathit{SetMap} \bullet \#\delta(\mathit{Unit}) = 1$$

If the set map is clear from the context we define *unit* by $\delta(\mathit{Unit}) = \{\mathit{unit}\}$.

Note that the *Unit* type variable can be used to define subsets and elements of a type as degenerate relations. For example a subset of the type variable Y would be the type $\mathit{Unit}* \rightarrow *Y$ and an element would be $\mathit{Unit}* \rightarrow! Y$.

4.22 Definition: Language Syntax

$$
\begin{array}{rr}
Exp ::= Exp + Exp & union \\
\mid Exp \;\&\; Exp & intersection \\
\mid Exp - Exp & difference \\
\mid Exp.Exp & navigation \\
\mid \sim Exp & transpose \\
\mid +Exp & closure \\
\mid \{ Var : TypeVar \mid Formula \} & set\ former \\
\mid Var &
\end{array}
$$

$$
\begin{array}{rr}
Formula ::= Exp\ in\ Exp & subset \\
\mid !Formula & negation \\
\mid Formula \;\&\&\; Formula & conjunction \\
\mid Formula \;||\; Formula & disjunction \\
\mid all\ Var : TypeVar \mid Formula & universal \\
\mid some\ Var : TypeVar \mid Formula & existential
\end{array}
$$

4.23 Definition: Type Judgement

The multiplicity markings, although relevant to the type semantics are superfluous for making type judgements, so we drop them. Given a type map Γ, we define the type judgement using natural deduction below.

Note, we write $\Gamma, v : Unit \to A$ for $\Gamma \oplus \{v \mapsto (Unit * \to! A)\}$. In the following A, B, C are any type variables, v is any variable, d, e are any expressions, F, G are any formulae.

$$\frac{}{\Gamma \vdash v : A \to B} \quad [\, \Gamma(v) = A\ m \to n\ B \,]$$

For any $cons \in \{+, \&, -\}$:

$$\frac{\Gamma \vdash d : A \to B,\ \Gamma \vdash e : A \to B}{\Gamma \vdash d\ cons\ e : A \to B}$$

$$\frac{\Gamma \vdash d : A \to B,\ \Gamma \vdash e : B \to C}{\Gamma \vdash d.e : A \to C}$$

$$\frac{\Gamma \vdash e : A \to B}{\Gamma \vdash \sim e : B \to A}$$

$$\frac{\Gamma \vdash e : A \to A}{\Gamma \vdash +e : A \to A}$$

$$\frac{\Gamma, v : \mathit{Unit} \to A \vdash F}{\Gamma \vdash \{v : A \mid F\} : \mathit{Unit} \to A}$$

$$\frac{\Gamma \vdash d : A \to B,\ \Gamma \vdash e : A \to B}{\Gamma \vdash d\ \mathit{in}\ e}$$

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

For any $op \in \{\&\&, ||\}$:

$$\frac{\Gamma \vdash F,\ \Gamma \vdash G}{\Gamma \vdash F\ op\ G}$$

For any $quant \in \{all, some\}$:

$$\frac{\Gamma, v : \mathit{Unit} \to A \vdash F}{\Gamma \vdash quant\ v : A \mid F}$$

We say an expression e is well-formed with respect to Γ provided there exists $A, B \in \mathit{TypeVar}$ such that $\Gamma \vdash e : A \to B$ can be derived. We say a formula F is well-formed with respect to Γ provided $\Gamma \vdash F$ can be derived.

4.24 Definition: Compatible Signature

A formula (resp. expression) and a signature $S = (\Upsilon, \Gamma)$ are compatible if the formula is well-formed with respect to Γ and any type variables introduced with the set former and quantification constructs in the formula are members of Υ.

4.25 Definition: Semantics

Given a signature, $S = (\Upsilon, \Gamma)$, and a formula or expression which are compatible, we can define the denotational semantics with respect to any instantiation, $I = (\delta, \eta)$, of the

signature.

$$
\begin{array}{l}
[\![v]\!]^{\eta}_{\delta} = \eta(v) \\
[\![d + e]\!]^{\eta}_{\delta} = [\![d]\!]^{\eta}_{\delta} \cup [\![e]\!]^{\eta}_{\delta} \\
[\![d \& e]\!]^{\eta}_{\delta} = [\![d]\!]^{\eta}_{\delta} \cap [\![e]\!]^{\eta}_{\delta} \\
[\![d - e]\!]^{\eta}_{\delta} = [\![d]\!]^{\eta}_{\delta} \setminus [\![e]\!]^{\eta}_{\delta} \\
[\![d.e]\!]^{\eta}_{\delta} = [\![d]\!]^{\eta}_{\delta} \fatsemi [\![e]\!]^{\eta}_{\delta} \\
[\![\sim e]\!]^{\eta}_{\delta} = ([\![e]\!]^{\eta}_{\delta})^{\sim} \\
[\![+ e]\!]^{\eta}_{\delta} = ([\![e]\!]^{\eta}_{\delta})^{+} \\
[\![v : T \mid F]\!]^{\eta}_{\delta} = \{x : \delta(T) \mid [\![F]\!]^{\eta \oplus \{v \mapsto \{(unit, x)\}\}}_{\delta} \bullet (unit, x)\} \\
[\![d \ in \ e]\!]^{\eta}_{\delta} \Leftrightarrow [\![d]\!]^{\eta}_{\delta} \subseteq [\![e]\!]^{\eta}_{\delta} \\
[\![!F]\!]^{\eta}_{\delta} \Leftrightarrow \neg \ [\![F]\!]^{\eta}_{\delta} \\
[\![F \ \&\& \ G]\!]^{\eta}_{\delta} \Leftrightarrow [\![F]\!]^{\eta}_{\delta} \wedge [\![G]\!]^{\eta}_{\delta} \\
[\![F \ || \ G]\!]^{\eta}_{\delta} \Leftrightarrow [\![F]\!]^{\eta}_{\delta} \vee [\![G]\!]^{\eta}_{\delta} \\
[\![all \ v : T \mid F]\!]^{\eta}_{\delta} \Leftrightarrow \forall \, x : \delta(T) \bullet [\![F]\!]^{\eta \oplus \{v \mapsto \{(unit, x)\}\}}_{\delta} \\
[\![some \ v : T \mid F]\!]^{\eta}_{\delta} \Leftrightarrow \exists \, x : \delta(T) \bullet [\![F]\!]^{\eta \oplus \{v \mapsto \{(unit, x)\}\}}_{\delta}
\end{array}
$$

4.26 Definition: Functions Relating an Instantiation and its Quotient

We will relate the denotational semantics with respect to the instantiation and the quotient instantiation with the following functions. Let $S = (\Upsilon, \Gamma)$ be a signature, $I = (\delta, \eta)$ an instantiation of it and $A \ m \rightarrow n \ B \in \mathit{TypeExp}$.

$$
\begin{array}{l}
K^{A \rightarrow B}_{S,I} : [\![A \ m \rightarrow n \ B]\!]_{\delta} \rightarrow [\![A \ m \rightarrow n \ B]\!]_{\overline{\delta}} \\
K^{A \rightarrow B}_{S,I}(r) == \{(a, b) : r \bullet ([a]_{\sim^{A}_{S,I}}, [b]_{\sim^{B}_{S,I}})\}
\end{array}
$$

$$
\begin{array}{l}
L^{A \rightarrow B}_{S,I} : [\![A \ m \rightarrow n \ B]\!]_{\overline{\delta}} \rightarrow [\![A \ m \rightarrow n \ B]\!]_{\delta} \\
L^{A \rightarrow B}_{S,I}(r) == \bigcup \{(a, b) : r \bullet a \times b\}
\end{array}
$$

Note that for any $r : [\![A \ m \rightarrow n \ B]\!]_{\overline{\delta}}$

$$K^{A \rightarrow B}_{S,I}(L^{A \rightarrow B}_{S,I}(r)) = r$$

4.27 Definition: Banned Constructs

The constructs in the language which will be *banned* for our small model theorem are the following (where $var : \mathit{Var}$):

$$
\begin{array}{l}
all \ var : X \ldots \\
some \ var : X \ldots \\
\{var : X \mid \ldots\}
\end{array}
$$

i.e. quantification or set formation using the type variable X.

4.28 Theorem: Instance and Quotient are Related

Let e be an expression not using banned constructs. Let F be a formula not using banned constructs. Let $S = (\Upsilon, \Gamma)$ be a compatible signature, $I = (\delta, \eta)$ an instantiation of it. Let $A, B \in \Upsilon$. We claim:

$$(\Gamma \vdash e : A \to B) \Rightarrow (L^{A \to B}_{S,I}([\![e]\!]^{\overline{\eta}}_{\overline{\delta}}) = [\![e]\!]^{\eta}_{\delta})$$

$$(\Gamma \vdash F) \Rightarrow \ ([\![F]\!]^{\eta}_{\delta} \Leftrightarrow [\![F]\!]^{\overline{\eta}}_{\overline{\delta}})$$

Proof: We proceed by structural induction over the expression or formula.

4.28.1 Case: *var*

Suppose $\Gamma \vdash var : A \to B$. We show $L^{A \to B}_{S,I}([\![var]\!]^{\overline{\eta}}_{\overline{\delta}}) = [\![var]\!]^{\eta}_{\delta}$ by showing inclusion in both directions.

Let $(a, b) : L^{A \to B}_{S,I}([\![var]\!]^{\overline{\eta}}_{\overline{\delta}})$. By the definition of $L^{A \to B}_{S,I}$ it follows that $(a, b) \in \bigcup\{(a', b') : \overline{\eta}(var) \bullet a' \times b'\}$. So choose $(a', b') : \overline{\eta}(var)$ such that $(a, b) \in a' \times b'$, and then choose $(a_0, b_0) : \eta(var)$ such that $[a_0]_{\sim^A_{S,I}} = a'$ and $[b_0]_{\sim^B_{S,I}} = b'$. It follows that $a_0 \sim^A_{S,I} a$ and $b_0 \sim^B_{S,I} b$. We know $(a_0, b_0) \in \eta(var)$, hence by the definition of $\sim^A_{S,I}$ in 4.12 we have $(a, b_0) \in \eta(var)$. Similarly it follows $(a, b) \in \eta(var)$. Since $\eta(var) = [\![var]\!]^{\eta}_{\delta}$ we have shown: $L^{A \to B}_{S,I}([\![var]\!]^{\overline{\eta}}_{\overline{\delta}}) \subseteq [\![var]\!]^{\eta}_{\delta}$.

Now let $(a, b) : [\![var]\!]^{\eta}_{\delta}$. Then $(a, b) \in \eta(var)$. It follows by the definition of quotient instantiation that $([a]_{\sim^A_{S,I}}, [b]_{\sim^B_{S,I}}) \in \overline{\eta}(var)$ Thus:

$$\begin{aligned}(a, b) &\in [a]_{\sim^A_{S,I}} \times [b]_{\sim^B_{S,I}} \\ &\subseteq \bigcup\{(a', b') : \overline{\eta}(var) \bullet a' \times b'\} \\ &= L^{A \to B}_{S,I}([\![var]\!]^{\overline{\eta}}_{\overline{\delta}})\end{aligned}$$

So we have shown $[\![var]\!]^{\eta}_{\delta} \subseteq L^{A \to B}_{S,I}([\![var]\!]^{\overline{\eta}}_{\overline{\delta}})$

4.28.2 Case: $d + e, d \& e, d - e$.

The proof for these three cases is very similar. In the following proof each of: $(+, \vee)$, $(\&, \wedge)$, $(-, \wedge \neg)$ can be substituted for $(\oplus, \otimes)$.

In this case we may suppose the following:

$$\Gamma \vdash d : A \to B \wedge \Gamma \vdash e : A \to B$$
$$L^{A \to B}_{S,I}([\![d]\!]^{\overline{\eta}}_{\overline{\delta}}) = [\![d]\!]^{\eta}_{\delta} \wedge L^{A \to B}_{S,I}([\![e]\!]^{\overline{\eta}}_{\overline{\delta}}) = [\![e]\!]^{\eta}_{\delta}$$

We show, $L^{A\to B}_{S,I},([\![d \oplus e]\!]^{\overline{\eta}}_{\overline{\delta}}) = [\![d \oplus e]\!]^{\eta}_{\delta}$ as follows:

$(a,b) \in L^{A\to B}_{S,I}([\![d \oplus e]\!]^{\overline{\eta}}_{\overline{\delta}})$
$\Leftrightarrow$
$\exists (a',b') : [\![d \oplus e]\!]^{\overline{\eta}}_{\overline{\delta}} \bullet a \in a' \wedge b \in b'$
$\Leftrightarrow$
$\exists (a',b') : \overline{\delta}(A) \times \overline{\delta}(B) \bullet a \in a' \wedge b \in b' \wedge$
$\quad ((a',b') \in [\![d]\!]^{\overline{\eta}}_{\overline{\delta}} \otimes (a',b') \in [\![d]\!]^{\overline{\eta}}_{\overline{\delta}})$
$\Leftrightarrow$
$(a,b) \in L^{A\to B}_{S,I}([\![d]\!]^{\overline{\eta}}_{\overline{\delta}}) \otimes (a,b) \in L^{A\to B}_{S,I}([\![e]\!]^{\overline{\eta}}_{\overline{\delta}})$
$\Leftrightarrow$
$(a,b) \in [\![d]\!]^{\eta}_{\delta} \otimes (a,b) \in [\![e]\!]^{\eta}_{\delta}$
$\Leftrightarrow$
$(a,b) \in [\![d \oplus e]\!]^{\eta}_{\delta}$

4.28.3 Case $d.e$

In this case we may suppose:

$\Gamma \vdash d : A \to B \wedge \Gamma \vdash e : B \to C$
$L^{A\to B}_{S,I}([\![d]\!]^{\overline{\eta}}_{\overline{\delta}}) = [\![d]\!]^{\eta}_{\delta} \wedge L^{A\to B}_{S,I}([\![e]\!]^{\overline{\eta}}_{\overline{\delta}}) = [\![e]\!]^{\eta}_{\delta}$

We show, $L^{A\to C}_{S,I}([\![d.e]\!]^{\overline{\eta}}_{\overline{\delta}}) = [\![d.e]\!]^{\eta}_{\delta}$ as follows:

$(a,c) \in L^{A\to C}_{S,I}([\![d.e]\!]^{\overline{\eta}}_{\overline{\delta}})$
$\Leftrightarrow$
$\exists (a',c') : [\![d.e]\!]^{\overline{\eta}}_{\overline{\delta}} \bullet a \in a' \wedge c \in c'$
$\Leftrightarrow$
$\exists\, a' : \overline{\delta}(A);\ b' : \overline{\delta}(B);\ c' : \overline{\delta}(C) \bullet a \in a' \wedge c \in c' \wedge$
$\quad (a',b') \in ([\![d]\!]^{\overline{\eta}}_{\overline{\delta}}) \wedge (b',c') \in ([\![e]\!]^{\overline{\eta}}_{\overline{\delta}})$
$\Leftrightarrow$
$\exists\, b : \delta(B) \bullet (a,b) \in [\![d]\!]^{\eta}_{\delta} \wedge (b,c) \in [\![e]\!]^{\eta}_{\delta}$
$\Leftrightarrow$
$(a,c) \in [\![d]\!]^{\eta}_{\delta} \fatsemi [\![e]\!]^{\eta}_{\delta}$
$\Leftrightarrow$
$(a,c) \in [\![d.e]\!]^{\eta}_{\delta}$

4.28.4 Case $\sim e$

In this case we may suppose the following:

$\Gamma \vdash e : B \to A$
$L^{B\to A}_{S,I}([\![e]\!]^{\overline{\eta}}_{\overline{\delta}}) = [\![e]\!]^{\eta}_{\delta}$

We show, $L^{A\to B}_{S,I}([\![\sim e]\!]^{\overline{\eta}}_{\overline{\delta}}) = [\![\sim e]\!]^{\eta}_{\delta}$ as follows:

$(a,b) \in L^{A\to B}_{S,I}([\![\sim e]\!]^{\overline{\eta}}_{\overline{\delta}})$
$\Leftrightarrow$
$\exists (a',b') : [\![\sim e]\!]^{\overline{\eta}}_{\overline{\delta}} \bullet a \in a' \wedge b \in b'$
$\Leftrightarrow$
$\exists (b',a') : [\![e]\!]^{\overline{\eta}}_{\overline{\delta}} \bullet a \in a' \wedge b \in b'$
$\Leftrightarrow$
$(b,a) \in [\![e]\!]^{\eta}_{\delta}$
$\Leftrightarrow$
$(a,b) \in [\![\sim e]\!]^{\eta}_{\delta}$

4.28.5 Cases: $\{v : Y \mid F\}$, *all* $v : Y \mid F$, *some* $v : Y \mid F$

Recall these are banned constructs if $Y = X$. We assume $Y \neq X$ and make the following definitions:

$\Gamma' == \Gamma \oplus \{v \mapsto (Unit * \to! \ Y)\}$
$S' = (\Upsilon, \Gamma')$

For any $y : \delta(Y)$ we define:

$\eta_y == \eta \oplus \{v \mapsto \{(unit, y)\}\}$
$I_y == (\delta, \eta_y)$

Let $\overline{I_y} = (\overline{\delta}, \overline{\eta_y})$ be the quotient instantiation wrt. S'. Because $Y \neq X$, $\sim^X_{S',I_y} = \sim^X_{S,I_y}$ so $\overline{I_y}$ is also the quotient instantiation wrt. S.

Recall *unit* is defined by $\delta(Unit) = \{unit\}$. In this context, $\overline{\delta}(Unit) = \{\{unit\}\}$. Recall also $^Y_{S,I_y}$ is the identity relation on $\delta(Y)$. It follows:

$\overline{\eta_y} = \overline{\eta} \oplus \{v \mapsto \{(\{unit\}, \{y\})\}\}$

Since $\Gamma' \vdash F$ we may suppose inductively:

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

Case $\{v : Y \mid F\}$.

We prove $L^{Unit \to Y}_{(S,I)}([\![\{v : Y \mid F\}]\!]^{\overline{\eta}}_{\overline{\delta}}) = [\![\{v : Y \mid F\}]\!]^{\eta}_{\delta}$ as follows:

$(unit, y) \in L^{Unit \to Y}_{(S,I)}([\![\{v : Y \mid F\}]\!]^{\overline{\eta}}_{\overline{\delta}})$
$\Leftrightarrow$
$(\{unit\}, \{y\}) \in [\![\{v : Y \mid F\}]\!]^{\overline{\eta}}_{\overline{\delta}}$
$\Leftrightarrow$
$[\![F]\!]^{\overline{\eta} \oplus \{v \mapsto \{(\{unit\}, \{y\})\}\}}_{\overline{\delta}}$
$\Leftrightarrow$
$[\![F]\!]^{\overline{\eta_y}}_{\overline{\delta}}$
$\Leftrightarrow$
$[\![F]\!]^{\eta_y}_{\delta}$
$\Leftrightarrow$
$(unit, y) \in [\![\{v : Y \mid F\}]\!]^{\eta}_{\delta}$

Case $all\ v : Y \mid F$.

$[\![all\ v : Y \mid F]\!]^{\overline{\eta}}_{\overline{\delta}}$
$\Leftrightarrow$
$\forall\, y : \overline{\delta}(Y) \bullet [\![F]\!]^{\overline{\eta} \oplus \{v \mapsto \{(\{unit\}, y)\}\}}_{\overline{\delta}}$
$\Leftrightarrow$
$\forall\, y : \delta(Y) \bullet [\![F]\!]^{\overline{\eta} \oplus \{v \mapsto \{(\{unit\}, \{y\})\}\}}_{\overline{\delta}}$
$\Leftrightarrow$
$\forall\, y : \delta(Y) \bullet [\![F]\!]^{\overline{\eta_y}}_{\overline{\delta}}$
$\Leftrightarrow$
$\forall\, y : \delta(Y) \bullet [\![F]\!]^{\eta_y}_{\delta}$
$\Leftrightarrow$
$[\![all\ v : Y \mid F]\!]^{\eta}_{\delta}$

Case $some\ v : Y \mid F$, similarly.

4.28.6 Case: *d in e*

We may suppose:

$\Gamma \vdash d : A \to B$
$\Gamma \vdash e : A \to B$

and inductively:

$L^{A \to B}_{(S,I)}([\![d]\!]^{\overline{\eta}}_{\overline{\delta}}) = [\![d]\!]^{\eta}_{\delta}$
$L^{A \to B}_{(S,I)}([\![e]\!]^{\overline{\eta}}_{\overline{\delta}}) = [\![e]\!]^{\eta}_{\delta}$

Applying $K^{A\to B}_{(S,I)}$ to both sides:

$$[\![d]\!]^{\overline{\eta}}_{\overline{\delta}} = K^{A\to B}_{(S,I)}([\![d]\!]^{\eta}_{\delta})$$
$$[\![e]\!]^{\overline{\eta}}_{\overline{\delta}} = K^{A\to B}_{(S,I)}([\![e]\!]^{\eta}_{\delta})$$

If follows easily from their definitions 4.26 that $K^{A\to B}_{(S,I)}$ and $L^{A\to B}_{(S,I)}$ are monotonic wrt. the subset ordering. Thus:

$$[\![d]\!]^{\eta}_{\delta} \subseteq [\![e]\!]^{\eta}_{\delta} \Leftrightarrow [\![d]\!]^{\overline{\eta}}_{\overline{\delta}} \subseteq [\![e]\!]^{\overline{\eta}}_{\overline{\delta}}$$

hence:

$$[\![d \ in \ e]\!]^{\eta}_{\delta} \Leftrightarrow [\![d \ in \ e]\!]^{\overline{\eta}}_{\overline{\delta}}$$

4.28.7 Case: &&, ||

This case is trivial. This concludes the proof.

4.29 Definition: Scope

Given a signature $S = (\Upsilon, \Gamma)$, a scope Θ for S is a total map from Υ to $\mathbb{N} \cup \{\infty\}$.

We say Θ is a finite scope if and only if $\infty \notin \operatorname{ran} \Theta$.

4.30 Definition: $Size_2$

Let S be a signature and, Θ a scope for S.

We define $Size_2(S, \Theta)$ exactly as in 4.16 replacing $\#\delta$ with Θ.

4.31 Definition: Consistent/Valid within Scope

Let $S = (\Upsilon, \Gamma)$ be a signature, Θ be a scope for S, and F a formula compatible with S.

We say F is consistent within Θ provided there exists an instantiation of S, $I = (\delta, \eta)$, such that:

$$\forall A : \operatorname{dom} \delta \bullet \#\delta(A) \leq \Theta(A)$$
$$[\![F]\!]^{\eta}_{\delta}$$

We say F is valid within Θ if and only if $!F$ is not consistent within Θ.

Note that if Θ is a finite scope, it can be decided (by a model finder) whether F is valid within Θ. We do not prove this here.

4.32 Theorem: Small Model Theorem

Claim: Let F be a formula not using banned constructs (4.27) and S be a compatible signature. Let Θ be a scope for S. Let $\overline{\Theta} = \Theta \oplus \{X \mapsto \mathit{Size}_2(S, \Theta)\}$. If F is valid within $\overline{\Theta}$ then F is valid within Θ.

Proof: Let F be a formula not using banned constructs and S be a compatible signature. Let Θ be a scope for S, and let $\overline{\Theta} = \Theta \oplus \{X \mapsto \mathit{Size}_2(S, \Theta)\}$. We will prove the equivalent statement: If $!F$ is consistent within Θ then $!F$ is consistent within $\overline{\Theta}$.

Suppose $!F$ is consistent within Θ. Let $I = (\delta, \eta)$ be an instance such that:

$$\forall\, A : \operatorname{dom} \delta \bullet \#\delta(A) \leq \Theta(A)$$
$$[\![!F]\!]^{\eta}_{\delta}$$

Let $\overline{I} = (\overline{\delta}, \overline{\eta})$ be the quotient instantiation of I defined in 4.19. It follows from 4.19, 4.17:

$$\#\overline{\delta}(X) = \#[]_{\sim^X_{S,I}} \leq \mathit{Size}(S, I) \leq \mathit{Size}_2(S, \Theta)$$
and so
$$\forall\, A : \operatorname{dom} \overline{\delta} \bullet \#\overline{\delta}(A) \leq \overline{\Theta}(A)$$

But, 4.28 tells us:

$$[\![!F]\!]^{\overline{\eta}}_{\overline{\delta}}$$

So we may conclude $!F$ is consistent within $\overline{\Theta}$, completing our proof.

4.33 Corollary: Generalisation through Skolemization

We can modify the preceding theorem, by adding a step to transform the formula $!F$ using negation normal form and Skolemization. This removes existential quantification in $!F$ and hence allows universal quantification in the original formula F, which was previously a banned construct. In other words, we arrive at a more general theorem which allows universal quantification with appropriately increased thresholds.

Only two levels of quantifier nesting can be accommodated in the this language, but arbitrary nesting follows immediately from proposed future work - see 5.1.

4.34 Corollary: Infinite Scope Checks

In the above theorem it is possible that $\Theta(X) = \infty$ whilst $\overline{\Theta}(X) < \infty$, and $\Theta(X)$ could be an infinite scope, when $\overline{\Theta}(X)$ is a finite scope. Since model finders need a finite scope normally, we have extended the class of problems (i.e. combination of formulae, signatures and scopes) that can be checked in a finite time.

5 Conclusions and Further Work

We have formally proved a small model theorem which applies to a general class of formulae. The small model theorem can reduce some infinite state problems to finite state problems by generating bounds on the size of type variable carrier sets. It may be that for some problems the results of our theorem coincide with known decidability results, but for most problems our theorem does not give rise to a decision procedure: its novel contribution is to cut-down the choice of possible scopes by eliminating some of them as redundant, and we hit this 'sweet spot' when bounds can be given for some type variables only as a function of the scope on others.

We have made a significant step toward the proposed research goal, however there are further areas to look at which we now outline.

5.1 Differences between our Language and the Alloy Kernel Language

Our small model theorem was based on a language which differs from the Alloy Kernel language (AKL). These differences need to be eliminated so we can generate a small model theorem of practical value and can leverage the Alloy Analyser model finder. We explain the differences and how we might eliminate them.

The type expression syntax in the AKL is defined [Jac02a]:

$$\begin{aligned} &\mathit{TypeExp}_{AKL} ::= \mathit{TypeVar} \rightarrow \mathit{TypeVar} \\ &\mid \mathit{TypeVar} \Rightarrow \mathit{TypeExp}_{AKL} \end{aligned}$$

where $A \rightarrow B$ means relation between A and B (just like $A * \rightarrow * B$ in our language) and $A \Rightarrow t$ means total function from A to t. The AKL syntax also has a corresponding construct for function application. This is the only difference between our language and the AKL.

5.1.1 Recursive Type Syntax

Developing our small model theorem was non-trivial and we chose to remove recursive type syntax constructs ($\mathit{TypeVar} \Rightarrow \mathit{TypeExp}_{AKL}$) on the first attempt, because this significantly simplified our theorem without making it vacuous. We now believe that a recursive type syntax can be easily accommodated using the the notion of *logical relations* in a similar manner to [LN00].

5.1.2 Multiplicity Markings

Our language and has multiplicity markings, whereas the Alloy Kernel language does not. Although our theorem is more general than required in this sense, proceeding without multiplicity markings has the same effect as assuming all markings are $*$ in the calculation of thresholds. This is undesirable because it would significantly increase thresholds (see 4.16) and therefore introduce *state explosion*.

Fortunately the full Alloy language has multiplicity markings so the information to reduce thresholds is available. During the Analyser's compilation into the kernel language though, these multiplicity markings are replaced by adding appropriate conjuncts to the formula. Unless this is the last step of the compilation this represents an implementation issue; we don't don't want to rewrite the whole compiler. Moreover, these additional conjuncts, although redundant when multiplicity markings are used, typically use banned constructs (4.27), which violate the conditions of our small model theorem.

We may hope to re-use the existing compiler in our solution as follows. If a problem in the full language is compiled in to the AKL, then by looking at the original formulation one should be able to infer appropriate multiplicity markings for variable declarations in the AKL. Based on this information one should be able to define patterns which match the (now redundant) additional conjuncts, so they can be removed.

In the worst case we might seek to exploit our results by lifting them to the full language, or even reworking our theorem to operate at the level of the full language. However, such an approach could be theoretically challenging, especially given our discussion at the beginning of section 4.

Overall though, further work is required to inspect the existing compiler implementation and discuss the best way forward with the compiler maintainer. It may be that the compiler code can be easily modified to meet our needs, and fortunately the existing implementation uses an open source license.

5.2 Thresholds on Multiple Type Variables

As we saw in the Birthday Book example (3.1), we can generate a threshold on $Date$ but only once a scope is given for $Name$ and $BirthdayBook$. Generally our theorem does not work as well as we would like when there is more than one type variable satisfying the conditions of the theorem. Usually, applying the theorem repeatedly does not generate an overall threshold for all the type variables, because the threshold for one type is a function of the scope of the other types. The resulting simultaneous equations do not usually have solutions. However, it seems sensible to imagine the existence of a more general small model theorem which could provide thresholds on multiple type variables simultaneously and we would seek this improved version as part of future work.

5.3 Relation to Data Independence

Whilst it is clear that the definition of data independence (2.4.1) applies to a wide range of systems, it is perhaps less/ clear that our small model theorem applies so widely. We believe (at least in the case of a single data independent type for now - see 5.2) that our theorem applies to the same systems and it is important to argue our theorem applies widely for it to be of value.

Unfortunately our belief appears difficult or perhaps impossible to formulate as a mathematical question; our assertion does not prescribe any particular translation between model checking problems and model finding problems, and there may be many 'reasonable' translations. Nevertheless it may be fruitful to explore this issue further. Existing work on translation between state-based and event-based formalisms (e.g. [Bol02]) could provide insight here.

5.4 Generalisation over Language

A final area we would like to look at (if time permits) is the generalisation of our results over the language used. The construction we have given seems generic and could be applied to model finding in other typed languages e.g. Z. Although, the Alloy Analyser seems to be the only model finder for typed languages (see 2.6), so this would be mainly of theoretical interest. The proofs we have given may become clearer if we have abstract away from the particularities of Alloy.

5.5 Plan of Work

In the introduction, we described the goal of the thesis: the extension of model finding technique for guaranteeing correctness from finite systems, to systems where infinite data independent types are also allowed. The following tasks are proposed for the completed thesis:

- Prove a small model theorem over a modified version of our language — one with the additional type construct of total functions (i.e. a recursive type syntax).
- Extend the small model theorem to give thresholds for multiple type variables.
- Modify or reuse Alloy Analyser compiler so that the full Alloy language is compiled into a Kernel language with multiplicity markings intact, rather than the existing Kernel language; then implement automatic threshold generation.
- Formalise the relationship between this small model theorem and data independence.
- Generalise the small model theorem to apply to a general class of languages.

- Use the small model theorem in a case study.

References

[Bol02] Christie Bolton. On the refinement of state-based and event-based models. D.Phil. Thesis, University of Oxford, 2002.

[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.

[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.

[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, 2002.

[JRB99] Ivar Jacobson Jame Rumbaugh and Grady Booch. *The Unified Modelling Langauge Reference Manual*. Addison-Wesley, 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.

[McC] William McCune. Mace 2.0 reference manual and guide.

[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.

[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.

[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.