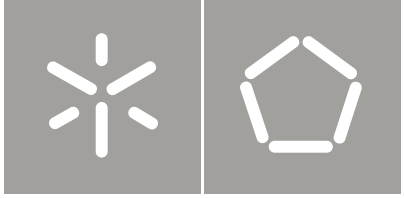
Universidade do Minho
Escola de Engenharia

Mário André Barbosa Eiras

Formalizing Alloy with a shallow embedding to Isabelle/HOL

Outubro de 2011

Formalizing Alloy with a shallow embedding to Isabelle/HOL
Mário André Barbosa Eiras
UMinho | 2011



Universidade do Minho
Escola de Engenharia

Mário André Barbosa Eiras

Formalizing Alloy with a shallow embedding to Isabelle/HOL

Tese de Mestrado
Engenharia Informática

Trabalho efectuado sob a orientação do
Professor Doutor Manuel Alcino Cunha

Outubro de 2011

Abstract

Formal methods are techniques developed with a mathematical basis in order to ensure a high level of quality on a software product. In this group of techniques there are some which favor the simplicity of use over the reliability of the results in order to reduce the resources that such approaches require. These so called "lightweight" formal methods emphasize partial specifications and rely on automatic analysis.

Alloy is a declarative specification language designed to be "lightweight". It was designed along with a model checking tool named Alloy Analyzer which can automatically analyze specifications and search for counter examples in a limited small scope. However, sometimes model checking is not enough and unbounded verification is needed.

In this work we defined a strategy to embed the logic of Alloy into the logic of the theorem prover Isabelle/HOL. We implemented a tool to automatically perform the shallow embedding, allowing the unbounded verification of Alloy specifications through the use of a theorem prover.

Resumo

Métodos formais são técnicas desenvolvidas com uma base matemática cujo objectivo é garantir um elevado nível de qualidade num produto de software. Entre este conjunto de técnicas existem algumas que privilegiam a simplicidade de uso sobre a fiabilidade dos resultados, a fim de reduzir os recursos que estas abordagens exigem. Estes métodos formais conhecidos como "leves", enfatizam a especificação parcial e análise automática.

Alloy é uma linguagem de especificação declarativa criada para ser "leve". Ela foi criada juntamente com uma ferramenta de model checking designada por Alloy Analyzer que pode analisar automaticamente as especificações e procura contra exemplos num pequeno universo. Contudo, por vezes uma abordagem como model checking não é suficiente e a verificação total é necessária.

Neste trabalho definimos uma estratégia para incorporar a lógica do Alloy na lógica do theorem prover Isabelle/HOL e implementamos uma ferramenta para executar automaticamente essa tradução, permitindo a verificação de especificações em Alloy através do uso de um theorem prover.

Contents

Chapter 1

Introduction

Nowadays, formal methods are widely used to specify software systems and verify them. Both these components are deeply connected since the verification step requires a specification. In the early stages of development of a software system, the developer knows the structure and the components that are supposed to be implemented and how these components should interact. The implementation of a software system comes later and reflects this abstract definition the developer had in mind at the start. Often, developers find out that the ideas they had about a software system where wrong, either because of some incoherence or in some extreme cases because of contradictory ideas. The worse is that in most cases these problems are found after its implementation.

The full formalization and verification of complex models can be a expensive task, both in time and money. The "lightweight formal methods" are a more viable way of verifying software. The idea is to specify and automatically analyze only those important and critical components of the software that justify the costs. By focusing the verification on a partial specification the time consumed in the specification process is reduced and by using automatic analysis the resources needed for verification are also significantly lower comparing with a theorem prover approach. Obviously, the verification

of a partial specification does not provide the same confidence level as the verification of a complete specification but on most cases it is enough.

One of those so called "lightweight" formal methods is Alloy [8]. Alloy is a formal language targeted at bounded model checking. It is based on Z notation [2], which in turn is based on first order logic and Zermelo-Fraenkel set theory.

The syntax is simple and resembles a typical object oriented language. It allows the specification of entities and the relationships between them; in fact, relations are the most important concept behind Alloy since they serve as the base for the specification of any model. It also allows the specification of invariants and propositions about entities and relations. This is done with relational algebra and first order logic, which grants Alloy great expressiveness but makes the verification of Alloy propositions an undecidable problem.

Generally there are various ways to verify that a model follows a given specification. The preferred way of doing this is through the use of automatic tools. Unfortunately this is not always possible because some specification languages, probably most, have an undecidable underlying logic. Alloy is in that category but, still, models can be automatically verified by the Alloy Analyzer. The truth is that this tool relies on bounded model checking algorithms to verify properties and does not prove the veracity of the specified assertions. Given a maximum size for the model, the Alloy Analyzer verifies the specified properties for all possible instances of the finite model. In that process, the model is translated to a propositional logic formula which can be verified by any SAT solver.

Although the Alloy language allows the definition of unbounded models, the Alloy Analyzer is limited to a finite scope specified by the user, and because of the exponential complexity of SAT solving algorithms this scope is very limited. So, even if the Alloy Analyzer proves some property P in a model of size n, it is always possible that P does not hold on a model of

size n + 1 or bigger. One possible solution is the use of theorem provers in the verification of Alloy models. Unlike model checking, theorem provers require user guidance to prove most properties, but they can be used to verify properties in arbitrarily big models.

Our goal is to develop a framework to perform unbounded verification of Alloy models using an off-the-shelf theorem prover. Theorem provers differ on the underlying logical system, being the most popular Higher-Order Logic (Isabelle [11] and PVS [12]) and the Calculus of Inductive Constructions (Coq [14]). Since Alloy logic (and syntax) is slightly different from those, an embedding of the model to the theorem prover logic must be performed.

Regarding embeddings, there are two possible approaches namely shallow embedding and deep embedding [16, 6]. In a shallow embedding the model is written in the theorem prover logic. For instance, a fact of an Alloy specification would translate to an axiom of the theorem prover. In a deep embedding the syntax of Alloy is translated to a data type which is then used to represent any specific model. A deep embedding allows reasoning over the structure of the model and language, but it is more difficult to understand and, therefore, verify the specification, whereas a shallow embedding is more readable and makes it easier to reason over the specification.

In this work we will describe the development and usage of our tool that automatically performs a shallow embedding of Alloy specifications into Isabelle/HOL. We will start with an overview of Alloy language in the chapter 2 where we present the semantics of a subset of the language. On chapter 3 we present an overview of Isabelle/HOL and on chapter 4 we analyze the two possible approaches regarding the embedding of Alloys's logic. On chapter 5 we explain how we perform the embedding of the logic of Alloy into Isabelle/HOL and on chapter 6 we explain how to verify an embedded specification in Isabelle/HOL. Chapter 7 presents an overview of some related work. On chapter 8 we present some conclusions about this work, exposing the main flaws and advantages of our approach.

Chapter 2

Alloy

Alloy [8] has been developed at MIT, appearing as a fairly limited prototype language for the first time in 1997. It has since passed trough a lot of changes, like syntactic tweaks and other major additions, that have increased the expressiveness of the language including the support of quantifiers and high arity relations. The Alloy language was developed together with the Alloy Analyzer tool to support the so called "lightweight formal methods". The language was designed to allow automatic verification and the Alloy Analyzer was created to fulfill that task.

Alloy is a simple declarative language capable of capturing the behavior and constraints of a software system. Alloy's underlying logic was heavily influenced by Z notation. Like Z, Alloy supports higher order quantifications although first order quantifications are the most used due to the limited support of higher order quantifications in the Alloy Analyzer. In Z, the most common datatypes are sets and relations which are also present in Alloy. In fact, all datatypes present in Alloy models are sets and relations but, in Alloy, a set is just a particular case of a relation and all relations are finite. Because of this, the concept of relation is an important one when dealing with Alloy. Despite of its influences in regards to the underlying logic, its syntax is actually very different from Z notation and is more reminiscent of

other languages such as the Object Constraint Language [13]. Being a simple language, Alloy is not expressive enough to capture every detail in a software system, but it is enough to capture the structural constraints and behavior of most systems and it has the advantage of allowing automatic analysis. In a few words, Alloy could be described as a language with a good balance between expressiveness and simplicity.

This chapter presents an overview of the Alloy language. Note that only the subset of the language important for this work is explained in this chapter. As an example we will construct a model of an address book similar to the one found in [8].

2.1 Relations

All data and structures in Alloy are represented as relations of arbitrary arity on a finite universe. There are three primitive relations, automatically defined in every Alloy specification:

- `univ` is the set of all atoms;
- `none` is the empty relation, usually denoted in common math notation as $\emptyset$;
- `iden` is the identity relation which can be described as the set of all tuples relating one atom to itself.

The most basic relations that can be created in Alloy are the unary relations which are sets of atoms. These relations are introduced with an signature as follows:

```
sig Name, Addr, Book { }
```

Here we have defined 3 relations. The relation `Name` which represents the set of names, the relation `Addr` which represents the set of addresses and the

relation `Book` representing the set of address books. These top-level relations are disjoint. In this specification, the relation `univ` is automatically defined as the union of the relations `Name`, `Addr` and `Book`.

A typical address book application may have the support for groups and the possibility to represent an address as an alias. Both, aliases and groups, are represented as names in an address book. Also, names and addresses can be the target of aliases and groups. To represent this we introduce the classification hierarchy of Alloy which is similar to the class hierarchy found in object oriented programing languages.

```
abstract sig Target {}
sig Addr extends Target {}
abstract sig Name extends Target {}
sig Alias, Group extends Name {}
sig Book {}
```

Now the relations `Name` and `Addr` are subsets of the new relation `Target`, and since this relation is declared as abstract it is in fact the union of the relations `Name` and `Addr`. The same can be said about the relation `Name` which is the union of the relations `Alias` and `Group`. The relations `Target` and `Book`, although not being explicitly defined as an extension of another relation, can be understood as an extension of `univ`, which in turn can be perceived as an abstract relation equal to the union of `Book` and `Target`.

Now that we have these sets of atoms we can specify the structure of the address book by declaring relations of arity higher than 1. Usually an address book has a set of names and for these names there should be some target.

```
sig Book {
    names: set Name,
    addr: names->some Target
}
```

In the context of the signature `Book`, the relation `names` is just a set of atoms from `Name` and the relation `addr` is a relation from `names` to `Target`. The words `set` and `some` indicate the multiplicity of the relations. Only two of these words are being used in this example, but there are four in total:

- `set` indicates that the multiplicity of the respective set in the relation can be anything, in fact we could think of it as the absence of multiplicity;
- `lone` indicates that there is no more than one atom from the respective set in the relation;
- `one` means there is exactly one atom related;
- `some` indicates that there is one or more elements related.

The relation `names` may look like a unary relation and `addr` may look like a binary relation, but they are respectively a binary and a ternary relation. In the case of `names` it relates each atom from `Book` with a set of atoms from `Name`, as for `addr` it relates each atom from `Book` with a binary relation between atoms from `names` and atoms from `Target`, where the multiplicity `some` indicates that for each atom from `names` there is at least one related atom in `Target`. Note that although `names` is a binary relation, in the context of the `Book` signature, it is interpreted as an unary relation.

2.2 Constraints

Now that we have specified the structure of our address book we need to add some constraints to specify its behavior, but for that purpose we need to have some knowledge about the relational operations and logic connectors of Alloy. The simpler logic expression is the relation inclusion, represented as `in`. Operators such as logic conjunction, represented as `or`, and logic

disjunction, represented as `and`, among others are part of Alloy and, since it is a first order logic, universal and existential quantifiers are also supported. Regarding relational operations, there are, among many other:

- `+` denoting the union of two relations with the same arity, yielding the set of all tuples from both relations.
- `^` denoting the transitive closure. Looking at a binary relation as a set of edges of a graph, the result of its transitive closure is the set of pairs that relate a vertice with all other vertices reachable from itself.
- `.` denoting the composition of relations. It has a similar meaning to the function composition. For instance, the composition of the relation $\{\langle \mathsf{a}, \mathsf{b}\rangle, \langle \mathsf{b}, \mathsf{c}\rangle\}$ with $\{\langle \mathsf{b}, \mathsf{b}, \mathsf{b}\rangle, \langle \mathsf{b}, \mathsf{c}, \mathsf{d}\rangle\}$ is $\{\langle \mathsf{a}, \mathsf{b}, \mathsf{b}\rangle, \langle \mathsf{a}, \mathsf{c}, \mathsf{d}\rangle\}$.

Now we are ready to define constraints in our model.

```
sig Book {
    names: set Name,
    addr: names->some (names + Addr)
}{
    no n: Name | n in n.^addr
    all a: Alias | lone a.addr
}
```

The first formula makes use of the transitive closure to specify that for any book there is no name related to itself neither directly nor indirectly, in other words, it means it is not possible to reach a name `n` from itself. The second constraint specifies that for every address book, any alias is related with a maximum of one address. We have also changed the declaration of the `addr` relation, so that a name of an address book is only related to addresses or names contained in the book.

The expected behavior of a software system is specified using assertions. For example, one obvious action that is performed on an address book is

search. A search action takes a book and a name and looks up the book for addresses related to that name. Let's begin by defining this function.

```
fun lookup [b: Book, n: Name] : set Addr { n.^(b.addr) & Addr }
```

The expression `n.^(b.addr)` is the set of targets related to `n`. Since we are only interested in addresses, the result is intersected with the set `Addr` in order to filter the relevant targets.

Finally, to specify how a search should behave, we make an assertion. If a name is present in an address book it should be related to some address. Using the previous declared function `lookup` we could specify this expected behavior:

```
assert lookupYields {
    all b: Book, n: b.names | some lookup [b,n]
}
```

2.3 Alloy Analyzer

Alloy Analyzer was the first tool designed to verify Alloy specifications and was created by the same people that created the Alloy language. Alloy was designed to be automatically checked for correctness so, to support the language, this tool was developed.

Alloy Analyzer checks the assertions of a given Alloy specification and tries to find a counter example in a bounded universe. If it finds a counter example it means that it is not valid, and it presents the found counter example as a graph where vertices represent atoms and the edges represent tuples of arities 2 and higher.

To exemplify, we define the operation `add` in address book specification. This operation adds a `Target` to a `Book`. In Alloy this can be done with a predicate stating that `Book b'` is the result of adding the `Target t` as a possible target for the `Name n` in the `Book b`.

```
pred add [b, b': Book, n: Name, t: Target] {
    b'.addr = b.addr + n->t
}
```

A behavior that some may be erroneously expecting is that if we add the `Target t` as a target for the `Name n` the targets related to a different `Name` do not change. This behavior is specified by the following assertion:

```
assert addLocal {
    all b, b': Book, n, n': Name, t: Target |
        add [b, b', n, t] and n != n'
        implies
        lookup [b, n'] = lookup [b', n']
}
```

To check this assertion we can use the command `check addLocal for 4 but 2 Book` which looks for counter examples in an universe with at most two atoms from `Book` and four atoms from `Target`. Figure 2.1 shows one of the counter examples found by Alloy Analyzer.

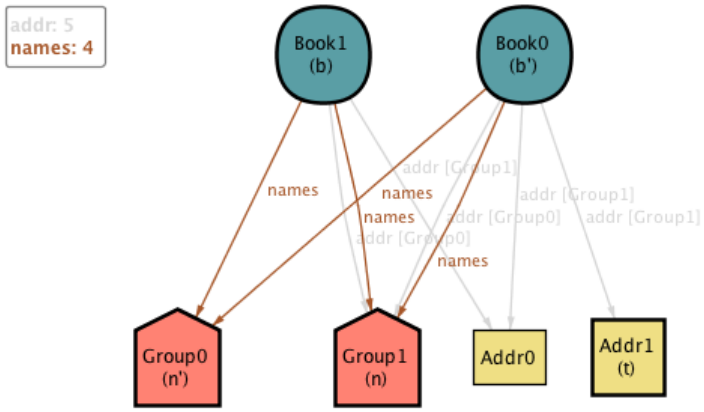


Figure 2.1: Counter example to the `addLocal` assertion.

In this figure, vertices represent books, groups and addresses. If a tuple `(b, n)` is in the relation `names` then it is represented as an arrow labeled `names` starting at the vertex b and pointing to the vertex n. Similarly, if a tuple `(b, g, t)` is in the relation `addr`, it is also presented as an edge but the label in this case is `addr[g]`. The counter example presented in the Figure 2.1 shows that if we add `Addr1` to `Group1` it will be reachable from `Group0` since this group is a target of `Group1`.

2.4 Grammar and Semantics

The goal of the address book example was to show some syntax and the purpose of an Alloy specification, but to fully understand the language we need to expand this knowledge further with a formal approach.

A specification in Alloy starts with an header indicating a name for the specification which may be followed by imports and then the paragraphs which are the definitions. The importance and the main objective of imports is to allow the usage of previously defined predicates and functions in other specifications without having to define them again. This is useful but unnecessary since the same result can be achieved through the declaration of the necessary elements instead of importing them. For this reason imports are ignored in this work but can be implemented in the future. Integers are also not supported. The complete supported grammar is presented in the Figure 2.2.

Figure 2.3 presents the semantics of relational expressions. In this figure ϕ denotes formulas, x denotes identifiers, A and B are denoting relational expressions, and the semantics of a relational expression A is denoted by $E[\![A]\!]_\Gamma$. We use Γ to denote the binding between the relation names and relations, which are just sets of tuples.

Some operations like `+` and `-` are equivalent to the respective set operations so their semantics is easy to describe. On the other hand, the semantics

module	::=	*header paragraph**
header	::=	`module` moduleId
paragraph	::=	*sigDecl* | *factDecl* | *funDecl* | *predDecl* | *assertDecl*
sigDecl	::=	[`abstract`] [*mult*] `sig` *sigID*,$^+$ [`extends` *sigRef*] *sigBody*
sigBody	::=	*decl*,* [*constraintSeq*]
factDecl	::=	`fact` [*factId*] *constraintSeq*
assertDecl	::=	`assert` [*assertId*] *constraintSeq*
funDecl	::=	`fun` [*sigRef* `::`] *funId* [*decl*,*] : *declExpr expr*
predDecl	::=	`pred` [*sigRef* `::`] *predId* [*decl*,*] *constraintSeq*
decl	::=	varId,$^+$: *declExpr*
letDecl	::=	varId `=` *expr*
declExpr	::=	*declSetExpr* | *declRelExpr*
declSetExpr	::=	[*mult*] *expr*
declRelExpr	::=	*declRelExpr* [*mult*] `->` [*mult*] *declRelExpr*
declRelExpr	::=	*declRelExpr* | *expr*
mult	::=	`lone` | `one` | `some`
expr	::=	[@] varId | *sigRef* | `this` | `none` | `univ` | `iden` | *unOp expr*
		| *expr binOp expr* | *expr* `[` *expr* `]`
		| `let` *letDecl*,$^+$ `|` *expr* | *constraint thenOp expr elseOp expr*
		| [*expr* ::] *funRef* `[` *expr*,* `]` | (*expr*)
constraintBody	::=	*constraintSeq* | `|` *constraint*
constraintSeq	::=	*constraint**
constraint	::=	*expr* [*neg*] *compOp expr* | *quantifier expr* | *neg constraint*
		| *constraint logicOp constraint* | *constraintSeq* | (*constraint*)
		| *constraint thenOp constraint* [*elseOp constraint*]
		| *quantifier decl*,$^+$ *constraintBody* | *expr* : *declExpr*
		| `let` *letDecl*,$^+$ *constraintBody* | [*expr* ::] *predRef* `[` *expr*,* `]`
thenOp	::=	`implies` | `=>`
elseOp	::=	`else`
neg	::=	`not` | `!`
logicOp	::=	`&&` | `||` | `iff` | `<=>` | `and` | `or`
quantifier	::=	`all` | `no` | *mult*
binOp	::=	`+` | `-` | `&` | `.` | `->` | `<:` | `:>` | `++`
unOp	::=	`~` | `*` | `^`
compOp	::=	`in` | `=`
funRef	::=	*funId*
predRef	::=	*predId*
sigRef	::=	*sigId* | `univ`

Figure 2.2: Alloy's grammar.

$$
\begin{aligned}
E[\![x]\!]_\Gamma &\equiv \Gamma(x) \\
E[\![A \texttt{ + } B]\!]_\Gamma &\equiv E[\![A]\!]_\Gamma \cup E[\![B]\!]_\Gamma \\
E[\![A \texttt{ - } B]\!]_\Gamma &\equiv E[\![A]\!]_\Gamma \setminus E[\![B]\!]_\Gamma \\
E[\![A \texttt{ \& } B]\!]_\Gamma &\equiv E[\![A]\!]_\Gamma \cap E[\![B]\!]_\Gamma \\
E[\![A \texttt{ . } B]\!]_\Gamma &\equiv \{\langle a_1,\cdots a_n, b_1,\cdots b_n\rangle \mid \\
&\qquad \exists x.\ \langle a_1,\cdots a_n, x\rangle \in E[\![A]\!]_\Gamma \wedge \\
&\qquad \langle x, b_1,\cdots b_n\rangle \in E[\![B]\!]_\Gamma\} \\
E[\![A \texttt{ :> } B]\!]_\Gamma &\equiv \{\langle a_1,\cdots a_n\rangle \mid \\
&\qquad \langle a_n\rangle \in E[\![B]\!]_\Gamma \ \wedge \langle a_1,\cdots a_n\rangle \ \in E[\![A]\!]_\Gamma\} \\
E[\![A \texttt{ <: } B]\!]_\Gamma &\equiv \{\langle a_1,\cdots a_n\rangle \mid \\
&\qquad \langle a_1\rangle \in E[\![A]\!]_\Gamma \ \wedge \langle a_1,\cdots a_n\rangle \ \in E[\![B]\!]_\Gamma\} \\
E[\![\texttt{let } x \texttt{ = } A \texttt{ | } B]\!]_\Gamma &\equiv E[\![B]\!]_{\Gamma \oplus (x \mapsto E[\![A]\!]_\Gamma)} \\
E[\![\phi \texttt{ => } A \texttt{ else } B]\!]_\Gamma &\equiv \begin{cases} E[\![A]\!]_\Gamma & \text{if } \phi \\ E[\![B]\!]_\Gamma & \text{if } \neg\phi \end{cases} \\
E[\![\texttt{\~{}} A]\!]_\Gamma &\equiv \{\langle a, b\rangle \mid \langle b, a\rangle \in E[\![A]\!]_\Gamma\} \\
E[\![\texttt{*}A]\!]_\Gamma &\equiv E[\![\texttt{\^{}}A + \texttt{iden}]\!]_\Gamma \\
E[\![\texttt{\^{}}A]\!]_\Gamma &\equiv \{\langle a, b\rangle \mid \langle a, b\rangle \in E[\![A]\!]_\Gamma \\
&\qquad \vee \exists x.\ \langle a, x\rangle \in E[\![A]\!]_\Gamma \wedge \langle x, b\rangle \in E[\![\texttt{\^{}}A]\!]_\Gamma\} \\
E[\![\texttt{univ}]\!]_\Gamma &\equiv \bigcup \{\Gamma(s) \mid s \text{ is a } \texttt{sig}\} \\
E[\![\texttt{iden}]\!]_\Gamma &\equiv \{\langle x, x\rangle \mid \langle x\rangle \in E[\![\texttt{univ}]\!]_\Gamma\} \\
E[\![\texttt{none}]\!]_\Gamma &\equiv \emptyset
\end{aligned}
$$

Figure 2.3: Semantics of relations.

of other operations, such as transitive closure, are more complex. Transitive closure and transitive reflexive closure are described using a recursive definition. If we look at a binary relation as a graph we can think of its transitive closure as the reachability relation where a is related with b if we can reach b from a. The relation `univ` is the relation that contains all atoms of the

specification, and for that reason it is defined as the union of all signature relations.

On figure 2.4 we describe the semantics of logic formulas. We use ϕ and ψ to denote formulas, x denotes identifiers, A and B denote relational expressions, t denotes tuples, and $F[\![\phi]\!]_\Gamma$ denotes the semantics of a formula ϕ. Although higher order quantifications are supported by the Alloy language they are rarely used because Alloy Analyzer does not support them, so we chose to support first order quantifications only. For this reason we define the semantics of this language using first order logic. Notice that $\Gamma \oplus (x \mapsto \{t\})$ binds the identifier x to the singleton relation $\{t\}$.

$$
\begin{aligned}
F[\![A \texttt{ in } B]\!]_\Gamma &\equiv E[\![A]\!]_\Gamma \subseteq E[\![B]\!]_\Gamma \\
F[\![A = B]\!]_\Gamma &\equiv E[\![A]\!]_\Gamma = E[\![B]\!]_\Gamma \\
F[\![\texttt{no } A]\!]_\Gamma &\equiv \neg F[\![\texttt{some } A]\!]_\Gamma \\
F[\![\texttt{lone } A]\!]_\Gamma &\equiv \forall x.\, \forall y.\, (x \in E[\![A]\!]_\Gamma \wedge y \in E[\![A]\!]_\Gamma) \implies x = y \\
F[\![\texttt{one } A]\!]_\Gamma &\equiv F[\![\texttt{lone} A]\!]_\Gamma \wedge F[\![\texttt{some} A]\!]_\Gamma \\
F[\![\texttt{some } A]\!]_\Gamma &\equiv \exists x. x \in E[\![A]\!]_\Gamma \\
F[\![\texttt{not } \phi]\!]_\Gamma &\equiv \neg F[\![\phi]\!]_\Gamma \\
F[\![\phi \texttt{ and } \psi]\!]_\Gamma &\equiv F[\![\phi]\!]_\Gamma \wedge F[\![\psi]\!]_\Gamma \\
F[\![\phi \texttt{ or } \psi]\!]_\Gamma &\equiv F[\![\phi]\!]_\Gamma \vee F[\![\psi]\!]_\Gamma \\
F[\![\phi \texttt{ iff } \psi]\!]_\Gamma &\equiv F[\![\phi]\!]_\Gamma \iff F[\![\psi]\!]_\Gamma \\
F[\![\texttt{no } x : A \mid \phi]\!]_\Gamma &\equiv \neg \exists t \in E[\![A]\!]_\Gamma.\, F[\![\phi]\!]_{\Gamma \oplus (x \mapsto \{t\})} \\
F[\![\texttt{lone } x : A \mid \phi]\!]_\Gamma &\equiv \forall t \in E[\![A]\!]_\Gamma.\, \forall t' \in E[\![A]\!]_\Gamma. \\
&\qquad (F[\![\phi]\!]_{\Gamma \oplus (x \mapsto \{t\})} \wedge F[\![\phi]\!]_{\Gamma \oplus (x \mapsto \{t'\})}) \implies t = t' \\
F[\![\texttt{one } x : A \mid \phi]\!]_\Gamma &\equiv F[\![\texttt{lone } x : A \mid \phi]\!]_\Gamma \wedge F[\![\texttt{some } x : A \mid \phi]\!]_\Gamma \\
F[\![\texttt{some } x : A \mid \phi]\!]_\Gamma &\equiv \exists t \in E[\![A]\!]_\Gamma.\, F[\![\phi]\!]_{\Gamma \oplus (x \mapsto \{t\})} \\
F[\![\texttt{all } x : A \mid \phi]\!]_\Gamma &\equiv \forall t \in E[\![A]\!]_\Gamma.\, F[\![\phi]\!]_{\Gamma \oplus (x \mapsto \{t\})}
\end{aligned}
$$

Figure 2.4: Semantics of formulas.

For such a simple language, Alloy actually has a complex type system. The most simple type rules are related to the arity of expressions. The operations `in`, `=`, `&`, `-` and `+` only apply to relations with the same arity, the composition can not be applied to two relations with arity 1, and the unary operations `~`, `*` and `^` only applies to binary relations. The type system also includes rules to detect irrelevant expressions. Basically the type of an expression is the product of the union of some signatures. The application of some operations may be irrelevant depending on the types of the relations. For instance, the intersection of two relations with the same arity but with disjoint types always yields `none`. The type system of Alloy Analyzer presents errors to the user when such expressions are found. This type system is presented with more detail in [15]. In this thesis we will assume all Alloy specifications to be type correct.

Chapter 3

Isabelle/HOL

Isabelle [11] is a generic theorem prover that provides a meta-logic allowing the instantiation of other logics. Probably, the most known logic used in Isabelle is HOL, which is an higher order logic instantiation allowing the specification of datatypes, inductive definitions and functions in addition to theorems. HOL is a typed logic, with a type system similar to those found in functional programing languages such as Haskell. In fact, the syntax itself shares some similarities with functional programing languages.

In this work's context, Isabelle was the theorem prover chosen to embed Alloy's logic. Isabelle/HOL provides a rich environment with enough expressiveness to embed Alloy's underling logic and therefore it is enough to fulfill our objectives. Besides there is no embedding of Alloy to Isabelle/HOL which makes this something new and useful for formal methods practitioners acquainted with Isabelle.

To better understand the embedding of Alloy's logic, this chapter presents an overview of Isabelle/HOL.

3.1 Types and terms

Isabelle/HOL is a typed logic with types ranging from the most simple like `bool`, the type of booleans, to more complex types with polymorphism. The possibilities to define a type are vast. The definition of the type `bool` for instance is a simple statement that this type exists. This type is then refined with constant definitions and axioms stating that it has exactly 2 habitants namely `True` and `False`. Another example is the polymorphic type of sets, defined as a type synonym of a function type.

```
types 'a set = "'a => bool"
```

Note that function types in Isabelle/HOL are represented using the binary type constructor `=>`. Also note that the type `"'a => bool"` is surrounded by quotation marks, which happens in Isabelle/HOL to make it possible to distinguish between the HOL specific terms and expressions from the meta logic. However, quotation marks can be avoided in simple expressions with only one term.

It is also possible to define completely new inductive data structures using a data type definition of the form:

$$\texttt{datatype}\ (\alpha_1, ..., \alpha_n)t = C_1\ \tau_{1_1}...\tau_{1_{k_1}} \mid ... \mid C_m\ \tau_{m_1}...\tau_{1_{k_m}}$$

where t is the type constructor, $\alpha_1, ..., \alpha_n$ are n different type variables, $C_1, ..., C_m$ are m different data constructors and τ_{i_j} is a type that may depend on the type variables. For instance, take the list data type definition as an example:

```
datatype 'a list =
    Nil    ("[]")
  | Cons 'a  "'a list"    (infixr "#" 65)
```

In this example the constructors are `Nil`, the empty list, and `Cons`, the constructor of non-empty lists. Note that for both constructors an alternative representation is being defined, `[]` as a synonym of `Nil` and `#` as an infix operator synonym of `Cons`. The constructor `Cons` takes an element of type `'a` as the head of the list and a list of type `"'a list"` as the tail. Also, a `datatype` definition automatically defines some laws useful to deal with the respective types in proofs. This includes an induction principle, which in this particular case is encoded in the following lemma:

```
[|?P []; !!a list. ?P list ==> ?P (a # list)|] ==> ?P ?list
```

This means that if a property P is valid for the empty list `[]` and if, for any `a` and `list`, being valid for `list` implies it is also valid for `a # list` then it must be valid for every list. In this lemma `!!` is the universal quantifier of the meta logic and `==>` is the implication of the meta logic. The symbol `?` indicates that the variable following it is an unknown that can be arbitrarily instantiated as needed.

Types and data structures alone do not have a great functionality or purpose. What is still missing are the functions. Functions in Isabelle/HOL are total, which means they always terminate. If the recursion is of a primitive kind, where each call simplifies the arguments, then Isabelle can automatically prove that the recursion always terminates. Those functions usually can be defined in the form:

`primrec` *name* :: *type (optional syntax)* `where` *equations*

Take the following function as an example.

```
primrec append :: "'a list => 'a list => 'a list"
(infixr "@" 65) where
    append_Nil:"[] @ ys = ys"
  | append_Cons: "(x#xs) @ ys = x # xs @ ys"
```

This function appends a list to another. Since the arguments are always simplified it is obvious that it terminates so it must be a total function, otherwise "`primrec`" could not be used. This also defines the rules `append_Nil` and `append_Cons` corresponding to each equation allowing them to be used in other contexts.

Other functions that do not use recursion at all can be declared simply as an abbreviation. An example of this kind of function is the set membership. As it was mentioned earlier, sets are defined as a synonym of a function. An element x is said to be in the set A if and only if $A(x)$ equals `True`, so $x \in A \equiv A(x)$.

```
definition member :: "'a => 'a set => bool" where
  mem_def: "member x A = A x"
```

This way, the actual definition of a set is hidden to the user.

3.2 Proofs

A standard proof in Isabelle/HOL is done by applying a sequence of rules to simplify our goal until it becomes trivial. Given a list, if we append an empty list to it, it surely results in the same list we had in the beginning. We can prove this lemma as follows:

```
lemma app_Nil2: "xs @ [] = xs"
apply (induct_tac xs)
apply (subst append_Nil)
apply (rule refl)
apply (subst append_Cons)
apply (erule ssubst)
apply (rule refl)
done
```

First we apply the inductive rule shown before which simplifies our initial goal into other two proof goals, one for the case `Nil` and one for the case of `Cons`.

```
goal (2 subgoals):
 1. [] @ [] = []
 2. !!a list. list @ [] = list ==> (a # list) @ [] = a # list
```

The first goal is made trivial and solved by the reflexivity rule after performing a substitution using the rule `append_Nil`. To simplify the second goal we need to perform a similar substitution but in this case using the rule `append_Cons`.

```
goal (1 subgoal):
 1. !!a list. list @ [] = list ==> a # list @ [] = a # list
```

The next steps are trivial. Performing a substitution using the premise, we get a trivial goal solved by the reflexivity rule.

The problem with this kind of proof is that, although it is easy to make, it is hard to understand. Isar [10] is a structured proof language that extends the apply-style proofs, making proofs human readable. Figure 3.1 presents a simplified grammar of Isar proofs.

Using Isar to produce the same proof presented earlier we have a proof that follows the same strategy as before but is now more human readable.

```
lemma append_Nil2: "xs @ [] = xs"
proof (induct xs)
    case Nil
    show "[] @ [] = []"
    proof (subst append_Nil)
        show "[] = []"..
    qed
next
```

$$
\begin{aligned}
\textit{proof} ::= & \ \texttt{proof}\ [\textit{method}]\ \textit{statement}^*\ \texttt{qed} \\
| & \ \texttt{by}\ \textit{method} \\
| & \ \texttt{..} \\
\textit{statement} ::= & \ \texttt{fix}\ \textit{variables} \\
| & \ \texttt{assume}\ \textit{propositions} \\
| & \ [\texttt{from}\ \textit{fact}^*]\ (\texttt{show} \mid \texttt{have})\ \textit{propositions}\ \textit{proof} \\
| & \ \texttt{case}\ \textit{label} \\
\textit{propositions} ::= & \ \textit{proposition} \\
| & \ \textit{proposition}\ \texttt{and}\ \textit{propositions} \\
\textit{proposition} ::= & \ [\textit{label}\texttt{:}]\ \textit{string} \\
\textit{fact} ::= & \ \textit{label}
\end{aligned}
$$

Figure 3.1: Simplified grammar of Isar proofs.

```
    case Cons
    fix a xs
    assume p: "xs @ [] = xs"
    from p show "(a # xs) @ [] = a # xs"
    proof (subst append_Cons)
        from p show "a # xs @ [] = a # xs"
        proof (rule ssubst)
            show "a # xs = a # xs"..
        qed
    qed
qed
```

The rules applied in this proof are actually the same used in the apply-style proof but the reader is able to understand the context in which the rules are being applied and what is the result. The `proof` command applies the inductive rule of list to `xs`. The application of a rule is optional and the main purpose of the `proof` command is to change the proof mode from

"prove" to "state". Basically in the "prove" mode, we can write apply-style proofs whereas the "state" mode allows us to write Isar statements.

In the "state" mode we can use the command `case` which starts a proof of one of the cases of the previously applied induction rule. The command `show` changes the mode back from "state" to "prove". To prove this case we use the command `proof` again applying a new rule and we end the proof of this case by trivially solving it with `...` Note that each `proof` command must end with `qed`. To introduce the second case of the proof we use the command `next` and the rest is similar to the first case.

The Isar proof is actually bigger than the apply-style proof, so it is important to note that Isar is not intended to shorten the proofs. It is intended to make the verification process easier and more readable in complex proofs. This particular lemma is so simple that it could be verified in two steps by applying induction and then the method `auto`. The method `auto` automatically simplifies the proof and tries to apply certain rules and in some simple cases like this it is almost always enough to solve the proof.

3.3 Locales

In Isabelle/HOL a locale is a parametric theory. The basis of a locale consists of a series of parameters and assumptions. Introducing a parameter in a locale is done with the keyword `fix` and assumptions are introduced with the keyword `assume`. These elements only exist in the context of the locale.

Most commands can be used in the context of the locale by adding `(in` *locale-name*`)` after the command name, this adds the declaration to the respective locale and new declarations may use elements already declared in the locale. The commands that can be used include `definition`, `primrec`, `lemma` and others.

A locale can be applied to constants making the assumptions and lemmas of the locale available in every context of Isabelle/HOL. This requires us to

prove that every assumption holds for the provided constants. The result is that every declaration in the locale becomes available in HOL, for the provided instantiation.

Chapter 4

Shallow vs Deep embedding

To verify an Alloy specification using Isabelle/HOL, we must perform an embedding of Alloy's logic to the logic of Isabelle/HOL. There are two possible approaches [16, 6], namely a deep embedding and a shallow embedding.

In a shallow embedding the logic of the theorem prover is used directly to describe the specification model, in other words, it maps the specification language syntax to the theorem prover syntax. For instance, if a logic connective P exists in the modeling language and a logic connective P' exists in the theorem prover language with the same semantics, then this connective P' is a possible result of the mapping for P on a shallow embedding.

On the other hand, following the deep embedding strategy, the specification language would be formalized as datatypes of the theorem prover logic and its semantics would be described in functions for those datatypes. So, each model of the specification language would be mapped to an instance of that datatype. This means that those datatypes and semantic functions would be a part of the resulting embedding, and they would remain the same for every model of a specification language.

The objective of this chapter is to explore both strategies and find out which one is the best to our purposes. To better understand the differences between these approaches we introduce a toy language based on Alloy and

perform both embeddings into Isabelle/HOL. In the context of this new logic a relation is a set of pairs, of a base set U. The following grammar defines this relational logic where atomic propositions are inclusion between relational expressions:

$$
\begin{aligned}
\mathit{formula} = \ & \mathit{relation}\ \texttt{in}\ \mathit{relation} \\
| \ & \texttt{not}\ \mathit{formula} \\
| \ & \mathit{formula}\ \texttt{and}\ \mathit{formula}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{relation} = \ & \mathit{identifier} \\
| \ & \mathit{relation}\ \texttt{+}\ \mathit{relation} \\
| \ & \mathit{relation}\ \texttt{-}\ \mathit{relation} \\
| \ & \mathit{relation}\ \texttt{.}\ \mathit{relation} \\
| \ & \texttt{\~{}}\ \mathit{relation} \\
| \ & \texttt{iden} \\
| \ & \texttt{none}
\end{aligned}
$$

The semantic of this language can be defined for a given a binding Γ which maps identifiers (relation names) to the corresponding relation. The semantic of the language is similar to the one defined in Figure 2.3 and Figure 2.4, and is omitted.

4.1 Shallow embedding

In a shallow embedding, the underlying logic of the theorem prover is used directly to describe the model. Each formula, data type or function on the model is represented as a formula, data type or function with the same semantic value. This may result in a model very similar to the original, depending on the theorem prover syntactic sugar and similarities to the modeling lan-

guage. The disadvantage is that the structure of the model is lost, hence it is impossible to reason over its structure.

To illustrate this approach let's take the previous defined language and perform an embedding to Isabelle/HOL. In the context of this toy language, a relation is a set of pairs, which is already defined in the HOL standard libraries and is denoted as `('a * 'b) set`. The content of the list is irrelevant and may be left undefined. All operations presented are part of the standard Isabelle/HOL libraries. The following rules show a possible shallow embedding where the translation of a relation A is denoted by $R[A]$ and the translation of a formula ϕ is denoted by $F[\phi]$.

$$
\begin{aligned}
R[x] &= x \\
R[A\texttt{+}B] &= R[A]\ \texttt{Un}\ R[B] \\
R[A\texttt{-}B] &= R[A]\ \texttt{-}\ R[B] \\
R[A\texttt{.}B] &= R[A]\ \texttt{O}\ R[B] \\
R[\texttt{\~{}}A] &= \texttt{converse}\ R[A] \\
R[\texttt{iden}] &= \texttt{Id} \\
R[\texttt{none}] &= \{\} \\
F[A\ \texttt{in}\ B] &= R[A]\ \texttt{<=}\ R[B] \\
F[\texttt{not}\ \phi] &= \texttt{\~{}}F[\phi] \\
F[\phi\ \texttt{and}\ \psi] &= F[\phi]\ \texttt{\&}\ F[\psi]
\end{aligned}
$$

Looking at the definition of this shallow embedding it is possible to note similarities in some operations which is a great advantage since it benefits the readability of the code. The differences between the model and the embedding could be reduced even further by using the syntactic sugar of Isabelle.

According to these rules, translating a lemma such as "$\texttt{\~{}\~{}}A\ \texttt{in}\ A$" to Isabelle/HOL leads to the lemma `"(converse (converse A)) <= A"` which

can be proved automatically:

```
lemma "converse (converse X) <= X"
by auto
```

4.2 Deep embedding

In a deep embedding, the modeling language is described trough data types. For our toy language these could be:

```
datatype aBinOp = APlus | AMinus | AComp

datatype arel = AVar ident
              | AConv arel
              | ABinOp aBinOp arel arel
              | AID
              | ANone

datatype aform = AIn arel arel
               | ANot aform
               | AAnd aform aform
```

The type *ident* is not very important but, since all the identifiers are mapped to an habitant of *ident* though an injective function, the domain of *ident* should be as big as the domain of identifiers. A possible approach would be to define *ident* as a synonym of *nat*.

These data types describe the structure of the grammar. This makes the embedding process a straightforward task consisting of a simple mapping of the model symbols to the data type symbols with some minor tweaks.

Since these data types alone do not have the semantic value of the original language, for the embedding to be complete, there must be an additional

definition. A formula on the original model is mapped to an habitant of this data type so the verification of assertions is only possible through the computation of its semantic value. As defined in the shallow embedding, the semantic value of a relation is a set of pairs and the semantic value of a formula is a boolean. In the following definitions, b is a binding that maps identifiers to relations, `R[b, x]` and `F[b, x]` denotes the semantic value of x in the enviroment `b`, of relational expressions and formulas, respectively.

```
primrec semr :: "(nat => ('a * 'a) set) => arel => ('a * 'a) set"
("R[_,_]" 200) where
  "R[b, AVar x]         = b x" |
  "R[b, AConv e]        = (converse (R[b,e]))" |
  "R[b, ABinOp bop l r] = (case bop of
                             APlus => R[b,l] Un R[b,r]
                           | AMinus => R[b,l] - R[b,r]
                           | AComp => (R[b,l]) O (R[b,r]))" |
  "R[b, ANone]          = {}" |
  "R[b, AID]            = Id"

primrec semf :: "(nat => ('a * 'a) set) => aform => bool"
("F[_,_]" 200) where
  "F[b, AIn l r]    = (R[b,l] <= R[b,r])" |
  "F[b, ANot f]     = (~ (F[b,f]))" |
  "F[b, AAnd l r]   = (F[b,l] & F[b,r])"
```

With this approach the expression "`~~`A `in` A" is translated to HOL as "`F[b, AIn (AConv (AConv (AVar 1))) (AVar 1)]`" where b is a well formed instanciation that maps the relation name "1" to a relation.

Since the structure of the data types is identical to the grammar, it means the structure of the model is preserved in the embedding which allow us to reason by induction over the structure of a formula or a relation and allow us to define properties about its structure. This is impossible in a shallow embedding since it does not preserve the structure of the original model.

The syntax resulting from this approach might be confusing at first because of how the relation names are presented and because there is a semantic value function which makes the verification process harder and with bigger proofs compared with a shallow embedding.

This embedding allows us to prove generic properties about the language which cannot be proved in a shallow embedding. For instance, we might want to prove that the semantics of expressions is the same after a simplification algorithm is applied. To ilustrate this example take a look at the following simplification rules:

$$\begin{array}{cc} R\texttt{.none} \rightsquigarrow \texttt{none} & \texttt{none}\,\texttt{.}\,R \rightsquigarrow \texttt{none} \\ R\texttt{.iden} \rightsquigarrow R & \texttt{iden}\,\texttt{.}\,R \rightsquigarrow R \\ R\ \texttt{+ none} \rightsquigarrow R & \texttt{none + }\ R \rightsquigarrow R \\ R\ \texttt{- none} \rightsquigarrow R & \texttt{none - }\ R \rightsquigarrow \texttt{none} \\ \texttt{\~{}\~{}}R \rightsquigarrow R & \end{array}$$

With these rules in mind, we can implement an algorithm that simplifies the syntactic tree.

```
primrec simp :: "arel => arel" where
  "simp (ABinOp bop l r) = (
    let l' = simp l;
        r' = simp r in
        case bop of
          APlus => if l' = ANone
                     then r'
                     else if r' = ANone
                            then l'
                            else ABinOp bop l' r'
        | AMinus => if l' = ANone
                      then ANone
                      else if r' = ANone
                             then l'
```

```
                          else ABinOp bop l' r'
      | AComp => if l' = ANone | r' = ANone
                  then ANone
                  else if l' = AID
                         then r'
                         else if r' = AID
                                then l'
                                else ABinOp bop l' r')" |
"simp (AConv r) = (
  let r' = simp r in
    if EX x. r' = AConv x
      then THE x. r' = AConv x
      else AConv r')" |
"simp (AVar x) = (AVar x)" |
"simp ANone = ANone" |
"simp AID = AID"
```

Now we are able to express and verify that this simplification algorithm keeps the semantics of the original expressions by writing and proving the following lemma:

```
lemma "R[b, r] = R[b, simp r]"
```

The proof of this lemma is presented in the appendix A. Since the shallow embedding does not preserve the structure of the language it does not allow the verification of a property like this.

4.3 Conclusions

Shallow and deep embeddings are different approaches for the formalization of specification languages, but both have their exclusive advantages. Since the deep embedding of a language preserves its structure, it is possible to prove properties which can not be verified in a shallow embedding. On other

hand, the result of a shallow embedding is simpler and easier to understand and in most cases, proofs are simpler than they would be on a deep embedding. The conclusion is that both approaches are good for different targets. In this work we are not interested in the properties of the language and we do not intend to verify properties about it. Our intentions are to verify Alloy specifications and for that reason a shallow embedding is the appropriate choice.

Chapter 5

Embedding Alloy into Isabelle/HOL

The embedding of an Alloy specification into Isabelle/HOL requires the translation of its syntactic constructs. Our goal is to define a semantic preserving embedding and implement a tool to perform it automatically. This chapter describes this process in detail.

5.1 Relations

This sections describes the theory `Alloy.thy` which includes the definitions of some types and relational operations. Relations of arbitrary arity are not explicitly defined in the standard Isabelle/HOL libraries so the first task is to define them. Since relations are sets of tuples and there is already a type `set` in Isabelle/HOL, part of the task is solved. However defining tuples of arbitrary arity is not so simple. Actually, types for tuples of any arity may be easily written in Isabelle/HOL using pairs, for instance triplets can be written as `('a * ('b * 'c))` but, these tuples have a problem. Since different arities are represented as different types, some relational operators require one definition for each possible arity. In the particular case of the

relational composition it would require one definition for each possible pair of arities and therefor the axiomatization of such a theory would be infinite. It is possible to use this typed approach (actually it was done before [1]) by defining only the operations needed in each specification. For instance, a model with binary and unary compositions only needs these two instances of composition defined. An automated tool could analyze the specifications and generate the definitions as needed. Since we assume that all Alloy models are type correct, there is a better option, which is to represent tuples using lists and allow relations to be arbitrary sets of such tuples. Type correctness will ensure that all tuples in a set will always have the same arity. With this untyped approach we only need one definition for each relational operation.

First we declare a type `atom`[1]. The habitants of this type are irrelevant and since we intend to analyze specifications for arbitrarily big domains, by defining this type we would be defining a bound for our verification. For those reasons it should be declared without any definitions:

```
typedecl atom
```

Next we define two type synonyms acording to what was explained before:

```
type_synonym tuple = "atom list"
type_synonym relation = "tuple set"
```

Variables are translated using a binding Γ that maps names to relations. In Alloy, quantified variables are singletons, but in HOL the quantified variables are tuples thus, every quantified variable v in the context of an expression is replaced with $\{v\}$. For every other relation name, the binding Γ returns a relation with the same name which is previously defined in HOL. In the figure 5.1 we present the translation of relational expressions.

Constants are translated to a relation with the the same name. In the particular case of the constants `univ`, `iden` and `none` we have the following definitions:

[1]Actually there is no need to define this type, instead we could use a type variable `'atom` but then the type `relation` would be a polymorphic type.

$$
\begin{aligned}
R[x]_\Gamma &\equiv \Gamma(x) \\
R[A \texttt{ + } B]_\Gamma &\equiv R[A]_\Gamma \texttt{ + } R[B]_\Gamma \\
R[A \texttt{ - } B]_\Gamma &\equiv R[A]_\Gamma \texttt{ - } R[B]_\Gamma \\
R[A \texttt{ .} B]_\Gamma &\equiv R[A]_\Gamma \texttt{ . } R[B]_\Gamma \\
R[A \texttt{ \& } B]_\Gamma &\equiv R[A]_\Gamma \texttt{ \& } R[B]_\Gamma \\
R[A \texttt{ :> } B]_\Gamma &\equiv R[A]_\Gamma \texttt{ :> } R[B]_\Gamma \\
R[A \texttt{ <: } B]_\Gamma &\equiv R[A]_\Gamma \texttt{ <: } R[B]_\Gamma \\
R[\phi \texttt{ => } A \texttt{ else } B]_\Gamma &\equiv \texttt{if } \phi \texttt{ then } R[A]_\Gamma \texttt{ else } R[B]_\Gamma \\
R[\texttt{let } x \texttt{ = } A \texttt{ | } B]_\Gamma &\equiv \texttt{let } x \texttt{ = } R[A]_\Gamma \texttt{ in } R[B]_\Gamma \\
R[*A]_\Gamma &\equiv *R[A]_\Gamma \\
R[\hat{}\, A]_\Gamma &\equiv \hat{}\, R[A]_\Gamma \\
R[\tilde{}\, A]_\Gamma &\equiv \tilde{}\, R[A]_\Gamma \\
R[\mathit{fun}\texttt{[}a_1, \ldots, a_n\texttt{]}]_\Gamma &\equiv \mathit{fun}\ R[a_1]_\Gamma \ \ldots\ R[a_n]_\Gamma \\
R[\texttt{univ}]_\Gamma &\equiv \texttt{univ} \\
R[\texttt{iden}]_\Gamma &\equiv \texttt{iden} \\
R[\texttt{none}]_\Gamma &\equiv \texttt{none}
\end{aligned}
$$

Figure 5.1: Embedding of relational expressions.

```
definition univ :: "relation" where
  "univ == {[x] | x. True}"

definition iden :: "relation" where
  "iden == {[x, x] | x. True}"

definition none :: "relation" where
  "none == {}"
```

The converse of some relation R is the set of the reversed lists (tuples) in that relation:

```
definition conv :: "relation => relation" ("~_" [999] 1000) where
```

```
    "~r == {rev x | x . x : r}"
```

The `&`, `+` and `-` operations are already defined in Isabelle/HOL and are written as `Int`, `Un` and `-` respectively, but since we want the embedding to be as similar as possible we defined equivalente operations with the syntax of Alloy.

```
abbreviation rel_union :: "relation => relation => relation"
(infixl "+" 65) where
    "r + s == r Un s"


abbreviation rel_inter :: "relation => relation => relation"
(infixl "&" 70) where
    "r & s == r Int s"
```

The composition of two relations A and B, is the set containing all the lists that can be created with lists from the set A and lists from the set B, for which the last atom in the list from A is the same as the first in the list from B. The lists with this correspondence, appended without the common element, form an element of the composed relation. It was defined as follows:

```
definition comp :: "relation => relation => relation"
(infixl "." 85) where
  "l . r == {xs @ ys | xs ys. EX w. (xs @ [w] : l) & (w # ys : r)}"
```

The product of two relations A and B is the the set of all lists created by appending one list from A with one list from B.

```
definition prod :: "relation => relation => relation"
(infixl "->" 75) where
  "prod a b == {x @ y | x y . (x : a) & (y : b)}"
```

The restriction operator `:>` filters the lists that start like some list on the restriction relation. Likewise the restriction operatior `<:` filters the lists that end like some list on the restriction relation.

```
definition doma :: "relation  => relation => relation"
(infixl "<:" 80) where
  "R <: S == {x . x : S & (EX y x' . y : R & x = y @ x')}"

definition rang :: "relation  => relation => relation"
(infixl ":>" 80) where
  "R :> S == {x . x : R & (EX y x' . y : S & x = x' @ y)}"
```

Probably the most complex operations in Alloy are the transitive closure and transitive-reflexive closure. These are defined using inductive set definitions. Any one of them could be defined by the other by a simple identity but our approach was to define both independently so the inductive rules are automaticaly defined for both operations as well.

```
inductive_set tcl :: "relation => relation" for r :: relation
where
  base: "[i, j] : r ==> [i, j] : tcl r" |
  step: "[i, j] : r ==> [j, k] : tcl r ==> [i, k] : tcl r"

abbreviation tcl2 :: "relation => relation" ("^_" [99] 99) where
    "^r == tcl r"

inductive_set trcl :: "relation => relation" for r :: relation
where
  base: "i : iden ==> i : trcl r" |
  step: "[i, j] : r ==> [j, k] : trcl r ==> [i, k] : trcl r"

abbreviation trcl2 :: "relation => relation" ("*_" [99] 99) where
    "*r == trcl r"
```

5.2 Formulas

Translating formulas from Alloy to Isabelle/HOL is, in part, a straightforward task. Most logical connectors and set predicates found in the semantics defined in figure 2.4 are already defined in Isabelle/HOL and therefore its translation is obvious.

Figure 5.2 shows how to embed Alloy formulas in Isabelle/HOL.

$$
\begin{aligned}
F[A \texttt{ in } B]_\Gamma &\equiv R[A]_\Gamma \texttt{ <= } R[B]_\Gamma \\
F[A\texttt{=}B]_\Gamma &\equiv R[A]_\Gamma \texttt{ = } R[B]_\Gamma \\
F[\texttt{no } A]_\Gamma &\equiv \texttt{no } R[A]_\Gamma \\
F[\texttt{lone } A]_\Gamma &\equiv \texttt{lone } R[A]_\Gamma \\
F[\texttt{one } A]_\Gamma &\equiv \texttt{one } R[A]_\Gamma \\
F[\texttt{some } A]_\Gamma &\equiv \texttt{some } R[A]_\Gamma \\
F[\texttt{not } \phi]_\Gamma &\equiv \texttt{\~{}} F[\phi]_\Gamma \\
F[\phi \texttt{ and } \psi]_\Gamma &\equiv F[\phi]_\Gamma \texttt{ \& } F[\psi]_\Gamma \\
F[\phi \texttt{ or } \psi]_\Gamma &\equiv F[\phi]_\Gamma \texttt{ | } F[\psi]_\Gamma \\
F[\phi \texttt{ iff } \psi]_\Gamma &\equiv F[\phi]_\Gamma \texttt{ <-> } F[\psi]_\Gamma \\
F[\texttt{no } x : A \mid \phi]_\Gamma &\equiv \texttt{\~{}} F[\texttt{some } A]_\Gamma \\
F[\texttt{lone } x : A \mid \phi]_\Gamma &\equiv \texttt{ALL x :} R[A]_\Gamma \texttt{. ALL y :} R[A]_\Gamma \texttt{.} \\
&\qquad (F[\phi]_{\Gamma \oplus (x \mapsto \{\texttt{x}\})} \texttt{ \& } F[\phi]_{\Gamma \oplus (x \mapsto \{\texttt{y}\})}) \texttt{ --> (x = y)} \\
F[\texttt{one } x : A \mid \phi]_\Gamma &\equiv F[\texttt{lone } x : A \mid \phi]_\Gamma \texttt{ \& } F[\texttt{some } x : A \mid \phi]_\Gamma \\
F[\texttt{some } x : A \mid \phi]_\Gamma &\equiv \texttt{EX x :} R[A]_\Gamma \texttt{.} \ F[\phi]_{\Gamma \oplus (x \mapsto \{\texttt{x}\})} \\
F[\texttt{all } x : A \mid \phi]_\Gamma &\equiv \texttt{ALL x :} R[A]_\Gamma \texttt{.} \ F[\phi]_{\Gamma \oplus (x \mapsto \{\texttt{x}\})} \\
F[\mathit{pred}\texttt{[}a_1, \ldots, a_n\texttt{]}]_\Gamma &\equiv \mathit{pred}\ R[a_1]_\Gamma \ \ldots \ R[a_n]_\Gamma
\end{aligned}
$$

Figure 5.2: Embedding of formulas.

Note that for some operations, instead of performing complex translations of formulas we chose to define equivalent operators in Isabelle/HOL with the same syntax used in Alloy. These operations are `no`, `lone`, `one` and `some`,

which are defined in the theory `Alloy.thy` using similar definitions to those found in the figure 2.4.

```
definition no :: "relation => bool" where
    "no R == R = {}"

definition lone :: "relation => bool" where
    "lone r == (ALL x y . ((x : r) & (y : r) --> (x = y)))"

definition one :: "relation => bool" where
    "one R == EX! x. x : R"

definition some :: "relation => bool" where
    "some R == EX x. x : R"
```

5.3 Declarations

A specifications in Alloy is translated to an Isabelle/HOL theory. To this theory we give the same name of the model. The theory `Alloy.thy` is always imported so the header looks like this:

`theory` *theory_name*
`imports Alloy`
`begin`

Our approach was to use a `locale` to define the content of a module. Although we did not perform the embedding of specifications with multiple modules, we chose to use a `locale` because it could make it easier in a future work. The definition of a `locale` uses the same name as the original Alloy module and all relations, facts, definitions and assertions in the module are defined in the context of the `locale`.

Signatures are the base of any Alloy model. Not only they define new relations but also properties regarding those relations. A signature alone declares a new unary relation and defines its hierarchy. A field declares a relation with a arity higher than 1 and defines some properties about its multiplicity. Consider the address book specification in the appendix B.1. The embedding of the declarations of this specification is presented in figure 5.3.

```
theory addressBook
imports Alloy
begin

locale addressBook =
    assumes fin_univ: "finite univ"
    fixes Target :: relation
        fixes Addr :: relation
        assumes Addr_def: "Addr in Target"
        fixes Name :: relation
            fixes Alias :: relation
            assumes Alias_def: "Alias in Name"
            fixes Group :: relation
            assumes Group_def: "Group in Name"
            assumes disj_Alias_Group: "disj Alias Group"
        assumes Name_def: "Name in Target"
        assumes abstract_Name: "Name = Alias + Group"
        assumes disj_Addr_Name: "disj Addr Name"
    assumes Target_def: "Target in univ"
    assumes abstract_Target: "Target = Addr + Name"
    fixes Book :: relation
        fixes names :: relation
        assumes Book_names_range: "ALL this : Book. (({this} . names) in Name)"
        fixes addr :: relation
        assumes Book_addr_range: "ALL this : Book.
            (({this} . addr) in (({this} . names) -> (({this} . names) + Addr)))"
        assumes Book_addr_mR1: "ALL this : Book. ALL var_0 : ({this} . names).
            some ({var_0} . ({this} . addr))"
    assumes Book_def: "Book in univ"
    assumes Book_fact_0: "ALL this : Book. ~(EX n : Name.
        ({n} in ({n} . ^({this} . addr)))) &
        (ALL a : Alias. lone ({a} . ({this} . addr)))"
    assumes disj_Target_Book: "disj Target Book"
    assumes names_type:"names in (Book -> Name)"
    assumes addr_type:"addr in (((Book -> Name) -> Name) + ((Book -> Name) -> Addr))"
```

Figure 5.3: Embedding of the signatures of the `addressBook` specification.

The first statement in the `locale` is the assumption that `univ` is finite. All relations and fields are declared with the `fixes` command followed with

the relation name. The most difficult task is to define the constraints of this relations. If a relations is abstract then it is the union of all its extensions so, in this example, `Name = Alias + Group`. If a relation is an extension of another relation, it is contained in its parent, for instance `Addr in Target`.

The constraints of a field declaration are far more complex than the others already explained. In this example, the first assumption regarding the field declaration `addr` simply states that for each book in `Book`, `addr` relates atoms from `({this} . names)` with atoms from `(({this} . names) + Addr)`. A similar assumption is present for every field declaration. The other assumption is related to the multiplicity of the relation `addr`, stating that each tuple from `({this} . names)` is related with some tuple from `({this} . addr)`. An assumption is generated for each multiplicity of the relation. For instance a relation declared as $R : S_1\ m_1\texttt{->}m_2\ \ldots\ m_{n-1}\texttt{->}m_n\ S_x$, where S_i are relations and m_i are multiplicities, generates n assumptions.

Generaly speaking, the translation of a signature and respective field declarations is done as follows:

D[`sig` A $\{R : S_1\ m_1\texttt{->}m_2\ \ldots\ m_{n-1}\texttt{->}m_n\ S_x\}$] =

`fixes` A
`fixes` R
`assumes "ALL this :` A`. ({this} .` R`) <=` S_1 `->` $\ldots$ `->` S_n`"`
`assumes "ALL this :` A`. ALL var_0 :` S_2`.` $\ldots$ `ALL var_`x`-2 :` S_x`.`
  m_1 `(((this .` R`) . var_`x`-2) . ... . var_0)`
`assumes "ALL this :` A`. ALL var_0 :` S_1`.` m_2 `(var_0 . (this .` R`))`
...
`assumes "ALL this :` A`. ALL var_0 :` S_x`.` m_{n-1} `(this .` R `. var_0)`
`assumes "ALL this :` A`. ALL var_0 :` S_1`.` $\ldots$ `ALL var_`x`-2 :` S_{x-1}`.`
  m_n `(var_`x`-2 . (... . var_0 . (this .` R`)))`

Additionally, if a signature A is abstract we add the assumption that A is the union of all its extensions. If A is not abstract then we write the assumption that A contains all its extensions.

Assertions are translated to lemmas with no proof. The proof must be written by the user. The Alloy commands `check` and `run` are completely ignored since their only purpose is to inform Alloy Analyzer the boundaries of verification. Functions and predicates are translated to definitions. Generally speaking, the translation of predicates, functions and assertions is done as follows:

P[**pred** *pred_name* [arg_1 : rel_1, ..., arg_n : rel_n] { *body* }] =
definition (in *locale_name*) *pred_name* :: "relation => ... => relation => bool" **where**
"*pred_name* arg_1 ... arg_n == F[*body*]"

P[**fun** *fun_name* [arg_1 : rel_1, ..., arg_n : rel_n] : *m rel* { *body* }] =
definition (in *locale_name*) *fun_name* :: "relation => ... => relation => relation" **where**
"*fun_name* arg_1 ... arg_n == R[*body*]"

A[**assert** *assert_name* { *body* }] =
lemma (in *locale_name*) *assert_name* : "F[*body*]"

Take a look at the Figure 5.4 for an example. This figure presents a predicate and an assertions from the address book specification found in the Appendix B.1.

```
definition (in addressBook) add ::
  "relation => relation => relation => relation => bool"
where
    "add b b' n t == ((b' . addr) = ((b . addr) + (n -> t)))"

lemma (in addressBook) delUndoesAdd:
"(ALL b : Book. ALL b' : Book. ALL b'' : Book. ALL n : Name. ALL t : Target.
   (no ({n} . ({b} . addr)) & (add {b} {b'} {n} {t}) & (del {b'} {b''} {n} {t})
     --> (({b} . addr) = ({b''} . addr))))"
```

Figure 5.4: Embedding of an assertion and a predicate of the addressBook specification.

5.4 Implementation

To preform this embedding process automatically we implemented a tool. The implementations was made in Java using some classes of Alloy Analyzer

which include the parser and lexical analyzer. The syntactic tree we get from the Alloy Analyzer methods is previously verified to be correctly or incorrectly typed. For this reason we assume that, if there is no error, the syntactic tree is well typed and no additional verification is needed.

The Java code can be found in `http://sourceforge.net/projects/alloytoisabelle` along with some examples.

The full embedding of the addressBook example automatically generated by this tool can be found in Appendix B.2.

Chapter 6

Verifying Alloy specifications using Isabelle/HOL

Verification of a specification is achieved by proving the generated lemmas. The basis for this verification is already defined in the libraries of Isabelle/HOL or in the provided `Alloy.thy` library. In `Alloy.thy` library we have included some lemmas that were helpful in the test cases we used. These are mostly simple generic properties such as associativity and distributivity of some operations.

6.1 Alloy auxiliary theory

There is no way to describe a strategy that always proves the required assertions, otherwise all this work would be pointless since it would be possible to automatically verify the assertions. The verification task requires us to look at what we assume to be true and somehow find a connection with the target property. In a big specification with a great number of facts this can be a hard task and in these cases it may help to start by selecting the assumptions that we can relate to the assertion we want to verify, and then trying to construct a proof focusing on those assumptions. A good strategy

to collect the facts we need is to use Alloy Analyzer and test if the assertion still holds when we remove some facts. This is not 100% effective but it is a great help. Writing the actual proof is more complicated because there is not much help one can get to consistently progress in this area. Practice is probably the best way to improve.

To help the verification process we have included some lemmas in `Alloy.thy`. These are generic lemmas such as associativity, transitivity, distributivity. The following are some of the approximately thirty lemmas included in this theory:

```
lemma prod_assoc:
  shows "A -> (B -> C) = (A -> B) -> C"

lemma prod_Un_dist:
  shows "A -> (B + C) = A -> B + A -> C"

lemma no_doma:
  assumes "R <= univ"
  shows "(R . S = {}) = (R <: S = {})"

lemma comp_distrib_l:
  shows "R . (S + T) = R . S + R . T"

lemma plus_minus:
  shows "no (R & S) ==> R + S - S = R"

lemma no_prod_in_comp:
  assumes "R <= univ"
  assumes "no (R . S)"
  assumes "X in R"
  shows "no (S & (X -> T))"
```

6.2 Arity

Because we have defined tuples as lists, sometimes we are required to prove that some relation has a specific arity. We can check the arity of a relation as follows:

```
definition arity :: "relation => nat => bool" where
    "arity r n == ALL x : r. n = length x"
```

We know that every relation that appears in our translated models has an arity but Isabelle/HOL does not know that because we chose to go with an untyped approach. The price to pay is that whenever we want to apply a lemma that only works for well formed relations or for a specific arity, we need to write a proof for that. For instance, the associativity of the composition depends on the arity of the middle relation:

```
lemma comp_assoc:
  assumes "arity B b"
  assumes "b > 1"
  shows "A . (B . C) = (A . B) . C"
```

Fortunately proving that a relation has an arity of n is simple but unfortunately it is also tedious. These proofs are done by repeatedly applying some simple lemmas.

```
lemma univ_arity:
shows "arity univ 1"

lemma iden_arity:
shows "arity iden 2"

lemma none_arity:
shows "arity none x"
```

```
lemma prod_arity:
shows "[|arity A a; arity B b|] ==> arity (A -> B) (a + b)"

lemma comp_arity:
shows "[|arity A a; arity B b|] ==> arity (A . B) (a + b - 1 - 1)"

lemma Un_arity:
shows "[|arity A n; arity B n|] ==> arity (A Un B) n"

lemma in_arity:
shows "[|arity A n; B <= A|] ==> arity B n"

lemma arity_tcl:
shows "arity  (^A) 2"


lemma arity_trcl:
shows "arity (*A) 2"
```

Although these lemmas are simple we did not find a way to use them automatically. In principle it should be possible to define a tactic to prove arity related properties automatically.

6.3 Examples of proofs

The first specification we verified was the address book example presented in the Chapter 2. Both the full specification and respective embedding can be found in the Appendix B. In this example we are required to verify the following assertion:

```
assert delUndoesAdd {
```

```
    all b, b', b'': Book, n: Name, t: Target |
        no n.(b.addr) and add [b, b', n, t] and del [b', b'', n, t]
        implies
        b.addr = b''.addr
}
```

This is a simple assertion stating that if we add a new tuple to a relation and then remove it, we have the original relation. This is obvious and easily verified by applying a similar lemma present in `Alloy.thy`. The first step in the proof is to apply the definitions of the predicates `add` and `del`. The next steps consist in the application of the two auxiliary lemmas `no_prod_in_comp` and `plus_minus`. We first show that `no ({b} . addr & {n} -> {t})` using the lemma `no_prod_in_comp` and then we use this property to prove the goal by using the lemma `plus_minus`.

```
lemma (in addressBook) delUndoesAdd:
"(ALL b : Book. ALL b' : Book. ALL b'' : Book. ALL n : Name. ALL t : Target.
    (no ({n} . ({b} . addr)) & (add {b} {b'} {n} {t}) & (del {b'} {b''} {n} {t}) -->
        (({b} . addr) = ({b''} . addr))))"
proof clarify
    fix b b' b'' n t
    assume b_in: "b : Book" and b'_in: "b' : Book" and b''_in: "b'' : Book"
        and n_in: "n : Name" and t_in: "t : Target" and no: "no ({n} . ({b} . addr))"
        and add: "add {b} {b'} {n} {t}" and del: "del {b'} {b''} {n} {t}"
    thus "{b} . addr = {b''} . addr"
    proof (simp add: add_def del_def)
        assume "n : Name" with Name_def and Target_def have n_in: "{n} in univ" by auto
        assume "no ({n} . ({b} . addr))" with n_in have "no ({b} . addr & {n} -> {t})"
        by (drule_tac no_prod_in_comp) auto
        with this show "{b} . addr = {b} . addr + {n} -> {t} - {n} -> {t}"
        by (drule_tac R = "{b} . addr" and S = "{n} -> {t}" in plus_minus) auto
    qed
qed
```

Another assertion in the address book specification is the following:

```
assert lookupYields {
    all b: Book, n: b.names | some lookup [b,n]
}
```

Which translates to the following Isabelle/HOL lemma:

```
definition (in addressBook) lookup ::
"relation => relation => relation" where
    "lookup b n == ((n . ^(b . addr)) & Addr)"

lemma (in addressBook) lookupYields:
    "(ALL b : Book. ALL n : ({b} . names). some (lookup {b} {n}))"
```

The definition of `lookup` uses transitive closure and in some cases this means the proof is not as straightforward as it is for other lemmas. If we think of the relation `b.addr` as the set of edges of a graph we can say that the `lookup` function gives the set of addresses reachable from the node `n`. So, this assertion states that for each name in a given book there is some addresses related to that name either directly or indirectly via another name. This is only true because the set of addresses is finite and that reflects in a proof that requires the set `Addr` to be finite. In this proof we use one of the most complex lemmas in the theory `Alloy.thy`:

```
lemma f_tcl:
assumes "N in univ"
assumes "L in univ"
assumes "finite N"
assumes "ALL n : N. EX x : N + L. n @ x : T"
assumes "~(EX n : N. {n} in {n} . ^T)"
shows "ALL n : N. EX l : L. (n @ l : ^T)"
```

This lemma states that if the set `N` is finite, if we can reach a vertex of `L` from every vertex in `N` and if there is no loop starting at the vertices of set `N`, then a vertex from `L` is reachable from a vertex in `N`.

The proof of `lookupYields` can be described in two parts. The first consists of showing that the assumptions of the lemma `f_tcl` hold for this particular context. More precisely it consists of showing that:

1. `{b} . name in univ`

2. `Addr in univ`

3. `finite ({b} . name)`

4. ```
   ALL n : {b} . name.
         EX x : {b}. name + Addr. n @ x : {b} . addr
   ```

5. `~(EX n : {b} . name. {n} in {n} . ^({b} . addr))`

The first three assumptions are simple and easy to verify. The fourth property is proved from the assumption `Book_addr_mR1` of the `locale` and the fifth is derived from a similar fact of the `Book` signature:

```
ALL this : Book.
    ~(EX n : Name. ({n} in ({n} . ^({this} . addr))))
```

The second part of the proof consists of showing that the result of the lemma `f_tcl` implies our goal. Since the goal is very similar to the property derived from `f_tcl`, this is a simple step.

The complete proofs can be found in Appendix B.2.
```

Chapter 7

Related work

Umbounded verification of Alloy specifications is not a new problem. Various embeddings of Alloy have been done before to various theorem prover logics. This chapter explores four alternatives.

7.1 Prioni

Prioni [1] is a tool that allows users to verify Alloy specifications using a theorem prover by performing embeddings of Alloy specification to the logic of a theorem prover. The target language of the embedding is Athena, a type-ω denotational proof language developed by Arkoudas, the author of Prioni, whose underlying logic is a polymorphic multi-sorted first order logic.

In Athena, types are introduced with a `domain` declaration such as:

```
(domain X)
```

It also supports polymorphic types, for instance the polymorphic type of sets may be introduced with the statement `(domain (Set-Of T))`. Athena also allows the definition of inductive types, generating an inductive rule and related axioms. Functions are declared with the command `declare`. Assuming that the types `A`, `B` and `C` are already defined, the statement `(declare`

`f (-> (A B) C))` declares a function `f` whose type is `A` $\rightarrow$ `B` $\rightarrow$ `C`. Note that in the declarations of polymorphic functions, the variables used in the polymorphic types must appear before the arrow:

```
(declare union ((T) -> ((Set-Of T) (Set-Of T)) (Set-Of T)))
```

Prioni translates Alloy specifications to a typed first-order finite-set theory providing an axiomization of Alloy's underlying logic and a library with some important lemmas which can be used in proofs. In the axiomization, relations are represented as sets of tuples. Relations are defined as needed. For instance, binary relations are defined as follows:

```
(structure (Pair-Of S T) (pair S T))
```

Since relations of different arities have different types, some relational operators must be defined for each relation arity. For instance, the composition of two binary relations is defined as follows:

```
(declare comp-2-2 ((S T U) -> ((Set-Of (Pair-Of S T))
  (Set-Of (Pair-Of T U))) (Set-Of (Pair-Of S U))))
(forall ?R1 ?R2 ?x ?y (iff (in (pair ?x ?y) (comp-2-2 ?R1 ?R2))
  (exists ?z (and (in (pair ?x ?z) ?R1)
    (in (pair ?z ?y) ?R2)))))
```

In a similar way to the definitions of relations, operators are also defined as needed so, if the composition of two relations with arity n and m appears in a specification, an operator `comp-`n`-`m is defined.

Transitive closure is defined using an exponentiation operation for binary relations. For instance some of the axioms are:

```
(define pow-def-1
  (forall ?R ?x ?y
    (iff (in (tup [?x ?y]) (pow ?R zero))
```

```
      (= ?x ?y))))

(define pow-def-2
  (forall ?R ?k ?x ?y
    (iff (in (tup [?x ?y]) (pow ?R (succ ?k)))
      (exists ?z
        (and (in [?x ?z] ?R)
          (in [?z ?y] (pow ?R ?k)))))))
```

The transitive closure is then defined with an axiom stating that if a pair (a, b) is in the transitive closure of R then there is a natural k such that (a, b) is in the power R^k

This implementation was done for one of the first versions of Alloy but, the language has evolved and today has some differences from what it was back then.

7.2 Dynamite

Another unbounded verification solution for Alloy language is Dynamite [5]. Like Prioni, the unbounded verification is done through an existing theorem prover (in this case PVS) and it also integrates Alloy Analyzer in a way that allows users to check the model in a finite small scope before trying to perform an unbounded verification with the theorem prover. The main differences between these two approaches are in the theory generated by the embedding. Whereas Prioni creates specific definitions and axioms for each arity present in the model relations, Dynamite converts all relations to binary relations, using the same finite axiomization for every model.

Dynamite embeddings result in a fork algebra theory defined by structures whose domain R is a set of binary relations of a universe B, in other words, $R \subseteq \mathcal{P}(B \times B)$.

Similar operators to those found in Alloy and some other are also part of

this algebra: union (+), intersection (&), complement (denoted as $\overline{r}$ for any binary relation r), converse ($\sim$), binary composition (.), reflexive-transitive closure ($*$), empty binary relation ($\emptyset$), universal bynary relation (denoted as 1) and the identity relation ($iden$). All these operators must be closed under B. There is a binary operation fork ($\triangledown$) defined as follows: $r \triangledown s = \{\langle a, b \star c\rangle \mid \langle a, b\rangle \in r \wedge \langle a, c\rangle \in s\}$

This definition requires an injective function $\star$ closed on B. Since $\star$ is injective, $a \star b$ can be seen as a representation of $\langle a, b\rangle$. The tuple projection relations π and ρ are defined as:

$$\pi = \{\langle a \star b, a\rangle \mid a, b \in B\}$$
$$\rho = \{\langle b \star a, a\rangle \mid a, b \in B\}$$

There is also the operation $\otimes$ defined as: $r \otimes s = \{\langle a \star b, c \star d\rangle \mid \langle a, c\rangle \in r \wedge \langle b, d\rangle \in s\}$

The composition of relations of different arities is handled differently due to the convertion of every relation to a binary relations. For any two relations R and S the composition $\bullet$ is handled as follows:

$$R \bullet S = \begin{cases} \sim \pi.Ran(R.S).\rho & \text{if } arity(R) = 1 \wedge arity(S) > 2 \\ Ran(R.S) & \text{if } arity(R) = 1 \wedge arity(S) \leq 2 \\ R.(iden \otimes (\cdots \otimes ((iden \otimes S).\pi))) & \text{if } |R| \neq 1 \text{ and } arity(S) = 1 \\ R.(iden \otimes (\cdots \otimes (iden \otimes S))) & \text{if } |R| \neq 1 \text{ and } arity(S) > 1 \end{cases}$$

With these operations defined, the translation of Alloy relations and expressions is done as follows:

$$
\begin{array}{ll}
T[C] = \mathrm{C} & F[r \text{ in } s] = T[r] \text{ in } T[s] \\
T[\sim r] = \sim T[r] & F[!\alpha] = !F[\alpha] \\
T[r.s] = T[r] \bullet T[s] & F[\alpha \mid\mid \beta] = F[\alpha] \mid\mid F[\beta] \\
T[*r] = *T[r] & F[\alpha \;\&\&\; \beta] = F[\alpha] \;\&\&\; F[\beta] \\
T[r + s] = T[r] + T[s] & F[some\ x : R \mid \alpha] = some\ x : T[R] \mid F[\alpha] \\
T[r - s] = T[r] \;\&\; \overline{T[s]} & F[all\ x : R \mid \alpha] = all\ x : T[R] \mid F[\alpha] \\
T[r \;\&\; s] = T[r] \;\&\; T[s] &
\end{array}
$$

With a pretty printer change in PVS, the translated formulas and relations are presented with a syntax similar to the Alloy language, which is a great advantage making it easier to understand. However the similarities disappear as proofs evolve. This approach seems to forget that Alloy relations are finite and some theorems may not be verifiable without this assumption.

7.3 Categorical calculus of relations

This work [3] is an evolution of the work described in the previous section [5]. It aims at embedding the logic of Alloy into a much simpler logic that in turn can be embedded in the logic of a theorem prover. Basically, it relies on the idea that a point free expression is simpler than its point wise version and therefore verifying an assertions in a point free version should be easier.

The embedding provided can be described by two major steps, the first part consists in embedding Alloy's logic into first order logic and the second part consists in embedding first order logic into categorical calculus of relations. The result is then automatically verified in Prover9 [9].

Relational logic is a first order logic enhanced with some relational operations which makes it an appropriate solution to represent Alloy's logic since they share many similarities. However, this representation is as complex as the original Alloy expressions. The excessive amount of quantifiers (and quantified variables) is not desired since it increases the complexity of

expressions so, in a second step every expression is translated to a point free version.

To solve the problem of translating relational logic formulas to point free formulas, this work explores the approach defined in [4] and defines a new solution that translates first order logic formulas to point free expression in a fork algebra (FA).

To simplify even further, an heuristic algorithm is used. This algorithm repeatedly applies simplification rules as long as it is possible.

The simplification of formulas is a major advantage for verification purposes but in most cases the meaning of the original specification becomes hard to understand, but since the purpose of this work is to use the automatic theorem prover Prover9 this is not relevant. Since the logic of Alloy is undecidable, this approach will not be able to verify some specifications, specially those with transitive closures.

7.4 SMT solving

This approach [7] consist of using a SAT modulo theories (SMT) solver to verify or refute assertions automatically. Alloy specifications are translated to the standard SMT-LIB language allowing them to be verified in multiple SMT solvers.

Relations declared by signatures and respective field declarations appearing in the specification are represented as a predicates in a first order logic. Hierarchical properties are also expressed trough predicates as well as multiplicity. Functions and predicates are eliminated by expanding every occurrence.

To test this approach the authors tried to verify some specifications that found in the library of examples of Alloy Analyzer. The solver verified the `com`[1] specification and others, but for the `markswep` specification the solver

[1]This specification is one of the examples presented in Alloy Analyzer.

was unable to verify any of its assertions. For comparison, we applied our approach to this example and succeeded in proving one of the assertions (see Appendix C). The source of complexity in this specification resides in the transitive closure. We also believe the solver is unable to verify the `lookupYields` assertion presented in the Appendix B, because it uses transitive closure and also because it is only true if a certain relation is finite. Unfortunately the assumption that every relation in Alloy is finite is lacking in this work.

The authors describe the marksweep specification as a complex one, but we actually think this problem is simpler then the COM assertions which they have verified automatically. This makes us conclude that there is still a long way to go for automatic approaches, mainly because of the difficult of dealing with transitive closure.

Chapter 8

Conclusion

We presented a solution to the problem of unbounded verification of Alloy. There is already some work on this subject as explained in Chapter 7. However there are some details that differentiate our approach from others. The most obvious difference is the target theorem prover, which is a great advantage for those who only know how to work with this Isabelle/HOL instead of PVS or Athena. Also, since automatic theorem provers cannot verify every assertion, we have the advantage of allowing to prove what the automatic approaches could not verify. For instance we were able to verify one assertion of the `marksweep` specification for which the SMT solver in [7] found false counter examples.

Our approach tries to create an embedding of Alloy into Isabelle/HOL that resembles Alloy as much as possible. This is a great advantage because it is easier for those familiar with Alloy to understand the specification in Isabelle/HOL. Unfortunately there are also one problem with our approach. We used an untyped approach because it allowed the definition of operations that work for any relation. The consequences are that in the context of a proof we might need to show that a relation has an arity n in order to apply a certain rule. This adds complexity to the proof without little benefit. As a future work we could try to find a workaround to this problem. One possi-

bility is implement tactics in Isabelle/HOL to solve these simple properties. Another approach to solve this problem would be to implement an automatic embedding to Coq such that relations are translated to an habitant of a dependent type. A dependent type would solve this problem by making the type of a relation dependent of its arity. This way the application of a lemma that depends on the arity of a relation could be applied without the need to prove that some relation has a particular arity.

Appendix A

Proof of a property in a deep embedding

```
lemma "R[b, r] = R[b, simp r]"
proof (induct r)
    case (AVar x) thus ?case by auto
next
    case (AConv x) thus ?case by (cases "simp x") auto
next
    case (ABinOp bop l r)
    assume l: "R[b,l] = R[b,simp l]" and r: "R[b,r] = R[b,simp r]"
    show ?case
    proof (cases bop)
        case APlus
        assume bop: "bop = APlus"
        show ?thesis
        proof (cases "simp l")
            case (AVar n) with l and r and bop show ?thesis by (cases "simp r") simp_all
        next
            case (AConv x) with l and r and bop show ?thesis by (cases "simp r") simp_all
        next
            case (ABinOp bop' l' r') with l and r and bop show ?thesis
                by (cases "simp r") simp_all
        next
            case AID  with l and r and bop show ?thesis by (cases "simp r") simp_all
        next
            case ANone with l and r and bop show ?thesis by (cases "simp r") simp_all
        qed
    next
```

```
        case AMinus
        assume bop: "bop = AMinus"
        show ?thesis
        proof (cases "simp l")
            case (AVar n) with l and r and bop show ?thesis by (cases "simp r") simp_all
        next
            case (AConv x) with l and r and bop show ?thesis by (cases "simp r") simp_all
        next
            case (ABinOp bop' l' r') with l and r and bop show ?thesis
                by (cases "simp r") simp_all
        next
            case AID  with l and r and bop show ?thesis by (cases "simp r") simp_all
        next
            case ANone with l and r and bop show ?thesis by (cases "simp r") simp_all
        qed
    next
        case AComp
        assume bop: "bop = AComp"
        show ?thesis
        proof (cases "simp l")
            case (AVar n) with l and r and bop show ?thesis by (cases "simp r") simp_all
        next
            case (AConv x) with l and r and bop show ?thesis by (cases "simp r") simp_all
        next
            case (ABinOp bop' l' r') with l and r and bop show ?thesis
                by (cases "simp r") simp_all
        next
            case AID  with l and r and bop show ?thesis by (cases "simp r") simp_all
        next
            case ANone with l and r and bop show ?thesis by (cases "simp r") simp_all
        qed
    qed
next
    case AID show ?case by auto
next
    case ANone show ?case by auto
qed
```

Appendix B

Address Book

B.1 Alloy specification

```
module addressBook

abstract sig Target { }
sig Addr extends Target { }
abstract sig Name extends Target { }

sig Alias, Group extends Name { }

sig Book {
    names: set Name,
    addr: names ->some (names + Addr)
} {
    no n: Name | n in n.^addr
    all a: Alias | lone a.addr
}

pred add [b, b': Book, n: Name, t: Target] { b'.addr = b.addr + n->t }
pred del [b, b': Book, n: Name, t: Target] { b'.addr = b.addr - n->t }
fun lookup [b: Book, n: Name] : set Addr { n.^(b.addr) & Addr }

assert delUndoesAdd {
    all b, b', b'': Book, n: Name, t: Target |
        no n.(b.addr) and add [b, b', n, t] and del [b', b'', n, t]
        implies
        b.addr = b''.addr
}
```

```
// This should not find any counterexample.
check delUndoesAdd for 3

assert addIdempotent {
    all b, b', b'': Book, n: Name, t: Target |
        add [b, b', n, t] and add [b', b'', n, t]
        implies
        b'.addr = b''.addr
}

// This should not find any counterexample.
check addIdempotent for 3

assert addLocal {
    all b, b': Book, n, n': Name, t: Target |
        add [b, b', n, t] and n != n'
        implies
        lookup [b, n'] = lookup [b', n']
}

// This shows a counterexample
check addLocal for 3 but 2 Book

assert lookupYields {
    all b: Book, n: b.names | some lookup [b,n]
}

// This should not find any counterexample.
check lookupYields for 4 but 1 Book
```

B.2 Embedding and proofs

```
theory addressBook
imports Alloy
begin

locale addressBook =
    assumes fin_univ: "finite univ"
    fixes Target :: relation
        fixes Addr :: relation
        assumes Addr_def: "Addr in Target"
        fixes Name :: relation
```

```
            fixes Alias :: relation
            assumes Alias_def: "Alias in Name"
            fixes Group :: relation
            assumes Group_def: "Group in Name"
            assumes disj_Alias_Group: "disj Alias Group"
        assumes Name_def: "Name in Target"
        assumes abstract_Name: "Name = Alias + Group"
        assumes disj_Addr_Name: "disj Addr Name"
    assumes Target_def: "Target in univ"
    assumes abstract_Target: "Target = Addr + Name"
    fixes Book :: relation
        fixes names :: relation
        assumes Book_names_range: "ALL this : Book. (({this} . names) in Name)"
        fixes addr :: relation
        assumes Book_addr_range: "ALL this : Book.
            (({this} . addr) in (({this} . names) -> (({this} . names) + Addr)))"
        assumes Book_addr_mR1: "ALL this : Book. ALL var_0 : ({this} . names).
            some ({var_0} . ({this} . addr))"
    assumes Book_def: "Book in univ"
    assumes Book_fact_0: "ALL this : Book.
        ~(EX n : Name. ({n} in ({n} . ^({this} . addr)))) &
        (ALL a : Alias. lone ({a} . ({this} . addr)))"
    assumes disj_Target_Book: "disj Target Book"
    assumes names_type:"names in (Book -> Name)"
    assumes addr_type:"addr in (((Book -> Name) -> Name) + ((Book -> Name) -> Addr))"

definition (in addressBook) add :: "relation => relation => relation => relation => bool"
where
    "add b b' n t == ((b' . addr) = ((b . addr) + (n -> t)))"

definition (in addressBook) del :: "relation => relation => relation => relation => bool"
where
    "del b b' n t == ((b' . addr) = ((b . addr) - (n -> t)))"

definition (in addressBook) lookup :: "relation => relation => relation" where
    "lookup b n == ((n . ^(b . addr)) & Addr)"

lemma (in addressBook) delUndoesAdd:
"(ALL b : Book. ALL b' : Book. ALL b'' : Book. ALL n : Name. ALL t : Target.
    (no ({n} . ({b} . addr)) & (add {b} {b'} {n} {t}) & (del {b'} {b''} {n} {t}) -->
        (({b} . addr) = ({b''} . addr))))"
proof clarify
    fix b b' b'' n t
    assume b_in: "b : Book" and b'_in: "b' : Book" and b''_in: "b'' : Book"
        and n_in: "n : Name" and t_in: "t : Target" and no: "no ({n} . ({b} . addr))"
        and add: "add {b} {b'} {n} {t}" and del: "del {b'} {b''} {n} {t}"
```

```
    thus "{b} . addr = {b''} . addr"
    proof (simp add: add_def del_def)
        assume "n : Name" with Name_def and Target_def have n_in: "{n} in univ" by auto
        assume "no ({n} . ({b} . addr))" with n_in have "no ({b} . addr & {n} -> {t})"
        by (drule_tac no_prod_in_comp) auto
        with this show "{b} . addr = {b} . addr + {n} -> {t} - {n} -> {t}"
        by (drule_tac R = "{b} . addr" and S = "{n} -> {t}" in plus_minus) auto
    qed
qed

lemma (in addressBook) addIdempotent:
"(ALL b : Book. ALL b' : Book. ALL b'' : Book. ALL n : Name. ALL t : Target.
    ((add {b} {b'} {n} {t}) & (add {b'} {b''} {n} {t}) -->
        (({b'} . addr) = ({b''} . addr))))"
by (auto simp add: add_def)

(* This assertion is false.
lemma (in addressBook) addLocal:
"(ALL b : Book. ALL b' : Book. ALL n : Name. ALL n' : Name. ALL t : Target.
    ((add {b} {b'} {n} {t}) & ({n} ~= {n'}) --> ((lookup {b} {n'}) = (lookup {b'} {n'}))))"
*)

lemma (in addressBook) lookupYields: "(ALL b : Book. ALL n : ({b} . names).
    some (lookup {b} {n}))"
proof (auto simp add: some_def lookup_def)
    fix b n
    from Name_def and Target_def have Name_in_univ: "Name in univ" by auto
    from this and fin_univ have finite_Name: "finite Name" by (rule finite_subset)
    assume b: "b : Book" with Book_names_range and Name_def
    have nin: "{b} . names in Name" by (simp add: Ball_def)
    with Target_def and Name_def
    have ns: "{b} . names in univ" by auto
    from b and Book_addr_mR1 have "ALL var_0 : ({b} . names).
        some ({var_0} . ({b} . addr))"
        by auto
    from this have a: "ALL n : ({b} . names). EX x : (({b} . names) + Addr).
        n @ x : ({b} . addr)"
    apply (auto simp only: some_def Ball_def)
    apply (drule_tac x = x in spec)
    apply (auto simp add: Bex_def)
    apply (rule_tac x = xa in exI)
    proof -
        fix x xa
        assume "x : {b} . names" with ns have U: "{x} in univ" by auto
        assume "xa : {x} . ({b} . addr)" then have I: "{xa} in {x} . ({b} . addr)"
            by auto
```

```
    from U and this have P: "x @ xa : {b} . addr"
    apply (drule_tac a = x and B = "{xa}" and C = "{b} . addr" in comp_to_prod)
        by (auto simp add: prod_def)
    assume "xa : {x} . ({b} . addr)" from this and P
    show "(xa : {b} . names | xa : Addr) && x @ xa : {b} . addr"
    proof auto
        assume "xa ~: Addr" and "xa : {x} . ({b} . addr)"
        from this and Book_addr_range and b have "xa : {b} . names"
        apply (simp add: Ball_def)
        apply (drule_tac x = "b" in spec)
        apply (drule_tac P = "b : Book" in mp)
        apply assumption
        apply (drule_tac C = "{x}" in comp_mono_l)
        apply (drule_tac A = "{x} . ({b} . addr)" and c = xa in subsetD)
        apply assumption
        apply (subgoal_tac "{b} in univ")
        apply (subgoal_tac "{x} in univ")
        apply (drule_tac X = "{xa}" and A = "{x}" and B = "{b} . names"
            and C = "{b} . names + Addr" in comp_prod)
        proof -
            from ns show "{b} . names in univ" by assumption
            assume "xa : {x} . ({b} . names -> ({b} . names + Addr))" then
            show "{xa} in {x} . ({b} . names -> ({b} . names + Addr))" by simp
            assume "{xa} in {b} . names + Addr" and "xa ~: Addr" then
                show "xa : {b} . names" by auto
        next
            from U show "{x} in univ" by assumption
        next
            assume "b : Book" with Book_def show "{b} in univ" by auto
        qed
        from this show "xa : {b} . names" by assumption
    qed
qed
have f: "ALL n : {b} . names. EX l : Addr. n @ l : tcl ({b} . addr)"
proof (rule_tac f_tcl)
    from ns show "{b} . names in univ" by assumption
    from Addr_def and Target_def show "Addr in univ" by auto
    from finite_Name and Book_names_range and b show "finite ({b} . names)"
        apply (simp add: Ball_def)
        apply (drule_tac x = "b" in spec)
        by (auto simp add: mp finite_subset)
    from a show "ALL n : {b} . names. EX x : {b} . names + Addr. n @ x : {b} . addr"
        by assumption
    from nin and Book_fact_0 and b
        show "~ (EX n : ({b} . names). {n} in {n} . ^({b} . addr))"
        by  (auto simp add: Ball_def)
```

```
    qed
    assume "n \<in> {b} . names"
    with b and f show "EX x. x : {n} . tcl ({b} . addr) & x : Addr"
    apply (simp only: Ball_def Bex_def)
    apply (drule_tac x = n in spec)
    apply (drule_tac mp)
    apply auto
    apply (rule_tac x = x in exI)
    proof -
        fix x
        assume "n @ x : tcl ({b} . addr)" then have aux: "{n} -> {x} in ^({b} . addr)"
            by (auto simp add: prod_def)
        assume "n : {b} . names" with nin have "{n} in Name" by auto
        with Name_in_univ have "{n} in univ" by auto
        with aux have aux: "x : {n} . ^({b} . addr)"
            apply (drule_tac B = "{x}" and C = "tcl ({b} . addr)" in prod_to_comp)
            by (auto simp add: some_def)
        assume "x : Addr" with aux show "x : {n} . ^({b} . addr) && x : Addr" by auto
    qed
qed

end
```

Appendix C

Mark and sweep garbage collection

C.1 Alloy specification

```
module marksweepgc

// a node in the heap
sig Node {}

sig HeapState {
  left, right : Node -> lone Node,
  marked : set Node,
  freeList : lone Node
}

pred clearMarks[hs, hs' : HeapState] {
  // clear marked set
  no hs'.marked
  // left and right fields are unchanged
  hs'.left = hs.left
  hs'.right = hs.right
}

// simulate the recursion of the mark() function using transitive closure
fun reachable[hs: HeapState, n: Node] : set Node {
  n + n.^(hs.left + hs.right)
}
```

```
pred mark[hs: HeapState, from : Node, hs': HeapState] {
  hs'.marked = hs.reachable[from]
  hs'.left = hs.left
  hs'.right = hs.right
}

// complete hack to simulate behavior of code to set freeList
pred setFreeList[hs, hs': HeapState] {
  // especially hackish
  hs'.freeList.*(hs'.left) in (Node - hs.marked)
  all n: Node |
    (n !in hs.marked) => {
      no hs'.right[n]
      hs'.left[n] in (hs'.freeList.*(hs'.left))
      n in hs'.freeList.*(hs'.left)
    } else {
      hs'.left[n] = hs.left[n]
      hs'.right[n] = hs.right[n]
    }
  hs'.marked = hs.marked
}

pred GC[hs: HeapState, root : Node, hs': HeapState] {
  some hs1, hs2: HeapState |
    hs.clearMarks[hs1] && hs1.mark[root, hs2] && hs2.setFreeList[hs']
}

assert Soundness1 {
  all h, h' : HeapState, root : Node |
    h.GC[root, h'] =>
      (all live : h.reachable[root] | {
        h'.left[live] = h.left[live]
        h'.right[live] = h.right[live]
      })
}

assert Soundness2 {
  all h, h' : HeapState, root : Node |
    h.GC[root, h'] =>
      no h'.reachable[root] & h'.reachable[h'.freeList]
}

assert Completeness {
  all h, h' : HeapState, root : Node |
    h.GC[root, h'] =>
```

```
      (Node - h'.reachable[root]) in h'.reachable[h'.freeList]
}

check Soundness1 for 3 expect 0
check Soundness2 for 3 expect 0
check Completeness for 3 expect 0
```

C.2 Embedding and proofs

```
theory marksweepgc
imports Alloy
begin

locale marksweepgc =
    assumes fin_univ: "finite univ"
    fixes Node :: relation
    assumes Node_def: "Node in univ"
    fixes HeapState :: relation
        fixes left :: relation
        assumes HeapState_left_range: "ALL this : HeapState.
            (({this} . left) in (Node -> Node))"
        assumes HeapState_left_mR1: "ALL this : HeapState. ALL var_0 : Node.
            lone ({var_0} . ({this} . left))"
        fixes right :: relation
        assumes HeapState_right_range: "ALL this : HeapState.
            (({this} . right) in (Node -> Node))"
        assumes HeapState_right_mR1: "ALL this : HeapState. ALL var_0 : Node.
            lone ({var_0} . ({this} . right))"
        fixes marked :: relation
        assumes HeapState_marked_range: "ALL this : HeapState.
            (({this} . marked) in Node)"
        fixes freeList :: relation
        assumes HeapState_freeList_range: "ALL this : HeapState.
            (({this} . freeList) in Node)"
        assumes HeapState_freeList_m: "ALL this : HeapState. lone ({this} . freeList)"
    assumes HeapState_def: "HeapState in univ"
    assumes disj_Node_HeapState: "disj Node HeapState"
    assumes marked_type:"marked in (HeapState -> Node)"
    assumes freeList_type:"freeList in (HeapState -> Node)"
    assumes left_type:"left in ((HeapState -> Node) -> Node)"
    assumes right_type:"right in ((HeapState -> Node) -> Node)"

definition (in marksweepgc) clearMarks :: "relation => relation => bool" where
    "clearMarks hs hs' ==
```

```
        no (hs' . marked) &
        ((hs' . left) = (hs . left)) & ((hs' . right) = (hs . right))"

definition (in marksweepgc) reachable :: "relation => relation => relation" where
    "reachable hs n == (n + (n . ^((hs . left) + (hs . right))))"

definition (in marksweepgc) mark :: "relation => relation => relation => bool" where
    "mark hs from hs' == ((hs' . marked) = (reachable hs from)) &
        ((hs' . left) = (hs . left)) & ((hs' . right) = (hs . right))"

definition (in marksweepgc) setFreeList :: "relation => relation => bool" where
    "setFreeList hs hs' == (((hs' . freeList) . *(hs' . left)) in (Node - (hs . marked)))
         & (ALL n : Node. (if ({n} !in (hs . marked))
        then no ({n} . (hs' . right)) &
            (({n} . (hs' . left)) in ((hs' . freeList) . *(hs' . left))) &
            ({n} in ((hs' . freeList) . *(hs' . left)))
        else (({n} . (hs' . left)) = ({n} . (hs . left))) &
            (({n} . (hs' . right)) = ({n} . (hs . right))))) &
            ((hs' . marked) = (hs . marked))"

definition (in marksweepgc) GC :: "relation => relation => relation => bool" where
    "GC hs root hs' == (EX hs1 : HeapState. EX hs2 : HeapState.
        (clearMarks hs {hs1}) & (mark {hs1} root {hs2}) & (setFreeList {hs2} hs'))"

lemma (in marksweepgc) Soundness1:
"(ALL h : HeapState. ALL h' : HeapState. ALL root : Node.
    ((GC {h} {root} {h'}) -->
    (ALL live : (reachable {h} {root}).
        (({live} . ({h'} . left)) = ({live} . ({h} . left))) &
        (({live} . ({h'} . right)) = ({live} . ({h} . right))))))"
proof (simp only: GC_def, clarify)
    fix h h' root hs1 hs2 live
    assume clearMarks: "clearMarks {h} {hs1}" and mark: "mark {hs1} {root} {hs2}"
        hence left: "{hs2} . left = {h} . left" by (simp add: clearMarks_def mark_def)
    from clearMarks and mark have right: "{hs2} . right = {h} . right"
        by (simp add: clearMarks_def mark_def)
    from left and right have reach: "reachable {h} {root} = reachable {hs2} {root}"
        by (simp only: reachable_def)
    from left have left: "{live} . ({hs2} . left) = {live} . ({h} . left)" by auto
    from right have right: "{live} . ({hs2} . right) = {live} . ({h} . right)" by auto
    from mark and reach have marked: "{hs2} . marked = reachable {h} {root}"
        by (simp add: mark_def reachable_def)
    assume "live \<in> reachable {h} {root}" with marked
        have live: "live : {hs2} . marked" by simp
    assume h: "h : HeapState" with HeapState_left_range
        have left': "{h} . left in Node -> Node" by simp
```

```
    with h and HeapState_right_range have right': "{h} . right in Node -> Node" by simp
    assume root: "root : Node" with Node_def have root': "{root} in univ" by auto
    assume "live : reachable {h} {root}" with root have live': "live : Node"
    proof (auto simp add: reachable_def)
        from left' and right' have "{h} . left + {h} . right in Node -> Node" by simp
        with Node_def have "^({h} . left + {h} . right) in Node -> Node"
            by (rule_tac tcl_in) auto
        hence p: "{root} . ^({h} . left + {h} . right) in {root} . (Node -> Node)"
            by (drule_tac C = "{root}" in comp_mono_l) simp
        assume "live : {root} . ^({h} . left + {h} . right)" with p
            have "{live} in {root} . (Node -> Node)" by auto
        with root' and Node_def show "live : Node"
            by (drule_tac X = "{live}" and B = "Node" and C = "Node" in comp_prod) auto
    qed
    assume "setFreeList {hs2} {h'}" with live' and live
            show "{live} . ({h'} . left) = {live} . ({h} . left) &&
                {live} . ({h'} . right) = {live} . ({h} . right)"
        apply (simp add: setFreeList_def Ball_def)
        apply (drule conjunct2)
        apply (drule conjunct1)
        apply (drule_tac x = live in spec)
        by (simp add: mp left right)
qed


lemma (in marksweepgc) Soundness2:
"(ALL h : HeapState. ALL h' : HeapState. ALL root : Node.
    ((GC {h} {root} {h'}) -->
        no ((reachable {h'} {root}) & (reachable {h'} ({h'} . freeList)))))"

lemma (in marksweepgc) Completeness:
"(ALL h : HeapState. ALL h' : HeapState. ALL root : Node.
    ((GC {h} {root} {h'}) -->
    ((Node - (reachable {h'} {root})) in (reachable {h'} ({h'} . freeList)))))"

end
```

Bibliography

[1] Konstantine Arkoudas, Sarfraz Khurshid, Darko Marinov, and Martin C. Rinard. Integrating model checking and theorem proving for relational reasoning. In Rudolf Berghammer, Bernhard Möller, and Georg Struth, editors, *RelMiCS*, volume 3051 of *Lecture Notes in Computer Science*, pages 21–33. Springer, 2003.

[2] S. M. Brien and J. E. Nicholls. *Z Base Standard*. ISO/IEC JTC1/SC22, 1992. Accepted for ISO standardization, ISO/IEC JTC1/SC22.

[3] Alcino Cunha and Nuno Macedo. Automated unbounded verification of alloy specifications with prover9. 2011.

[4] Marcelo F. Frias, Carlos López Pombo, and Nazareno Aguirre. An equational calculus for Alloy. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *ICFEM*, volume 3308 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2004.

[5] Marcelo F. Frias, Carlos López Pombo, and Mariano M. Moscato. Alloy Analyzer+PVS in the analysis and verification of Alloy specifications. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 587–601. Springer, 2007.

[6] François Garillot and Benjamin Werner. Simple types in type theory: Deep and shallow encodings. In Klaus Schneider and Jens Brandt, editors, *TPHOLs*, volume 4732 of *Lecture Notes in Computer Science*, pages 368–382. Springer, 2007.

[7] Aboubakr Achraf El Ghazi and Mana Taghdiri. Relational reasoning via SMT solving. In *17th International Symposium on Formal Methods (FM)*, pages 133–148, June 2011.

[8] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., 2006.

[9] W. McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[10] Tobias Nipkow. A tutorial introduction to structured isar proofs.

[11] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[12] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 2001.

[13] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[14] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.3*, April 2011. `http://coq.inria.fr`.

[15] Emina Torlak. *A constraint solver for software engineering: finding models and cores of large relational specifications*. PhD thesis, Cambridge, MA, USA, 2009. AAI0821754.

[16] Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *TPHOLs*, volume 3223 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2004.