Universidade do Minho
Escola de Engenharia

Nuno Filipe Moreira Macedo

Translating Alloy specifications to the point-free style

Setembro de 2010

Universidade do Minho
Escola de Engenharia

Nuno Filipe Moreira Macedo

Translating Alloy specifications to the point-free style

Mestrado em Informática

Trabalho efectuado sob a orientação do
Professor Doutor Manuel Alcino Cunha

Setembro de 2010

Acknowledgments

I would like to thank my supervisor, professor Alcino Cunha, for the guidance, encouragement and advice, for all the time spent improving the thesis, and most of all, for arousing my interest in formal methods.

I would also like to thank professor José Nuno Oliveira, for the inspiration and for providing valuable insight during the realization of this project.

My colleagues, Bruno, Jota, Mané, Miguel, Pacheco, Macedo and Vicente, for making working in a lab bearable.

My friends, Maurício, Joana, Costa, Jipi, Joca, Jó, Zé, Paulo, Victor and André, for keeping me sane outside the lab.

Eduarda, for simply being there in every step of the way.

My family, my grandparents, mom, dad, my brother, without their support I would not be able achieve this. To my grandfather, who isn't around to see me receive the diploma, I dedicate this thesis.

Abstract

Every program starts from a model, an abstraction, which is iteratively refined until we reach the final result, the implementation. However, at the end, one must ask: does the final program resemble in anyway the original model? Was the original idea correct to begin with? Formal methods guarantee that those questions are answered positively, resorting to mathematical techniques. In particular, in this thesis we are interested on the second factor: *verification of formal models*.

A trend of formal methods defends that they should be lightweight, resulting in a reduced complexity of the specification, and automated analysis. *Alloy* was proposed as a solution for this problem. In Alloy, the structures are described using a simple mathematical notation: *relational logic*. A tool for model checking, automatic verification within a given scope, is also provided.

However, sometimes model checking is not enough and the need arises to perform *unbounded* verifications. The only way to do this is to mathematically prove that the specifications are correct. As such, there is the need to find a mathematical logic expressive enough to be able to represent the specifications, while still being sufficiently understandable.

We see the *point-free* style, a style where there are no variables or quantifications, as a kind of *Laplace transform*, where complex problems are made simple. Being Alloy completely relational, we believe that a point-free relational logic is the natural framework to reason about Alloy specifications.

Our goal is to present a translation from Alloy specifications to a point-free relational calculus, which can then be mathematically proven, either resorting to proof assistants or to manual proving. Since our motivation for the use of point-free is simplicity, we will focus on obtaining expressions that are simple enough for manipulation and proofs about them.

Resumo

Todos os programas partem de um modelo, uma abstracção, que é iterativamente refinada até se atingir o resultado final, a implementação. No entanto, no fim, não há garantia de a implementação representar o modelo original. Nem sequer sabemos se o modelo inicial estava correcto. Métodos formais garantem que estas questões são respondidas positivamente. Nesta tese em particular estamos interessados no segundo factor: *verificação de modelos formais*.

Uma linha de pensamento nos métodos formais defende que estes devem ser *leves*, resultando numa diminuição da complexidade das especificações e na automação da sua análise. O *Alloy* foi proposto como uma solução para este problema. Em Alloy, as estruturas são definidas usando uma notação matemática simples: lógica relacional. Uma ferramenta para verificação de modelos, dentro de um dado limite, é também disponibilizada.

No entanto, às vezes essa verificação não é suficiente, e surge a necessidade de efectuar verificações *sem limites*. A única maneira de fazer isto é provando matematicamente que as especificações estão correctas. Sendo assim, surge a necessidade de encontrar uma lógica matemática que seja suficientemente expressiva para representar as especificações, mais ainda suficientemente compreensível.

Vemos o estilo "*point-free*", um estilo de programação sem variáveis, como uma espécie de transformada de Laplace, onde problemas complexos se tornam mais simples. Visto o Alloy ser completamente relacional, acreditamos que um ambiente "point-free" relacional é o mais natural para lidar com as suas especificações.

O nosso objectivo é apresentar uma tradução de especificações Alloy para lógica relacional "point-free", onde podem ser matematicamente provadas, quer recorrendo a assistentes de prova, quer manualmente. Visto a nossa motivação para o uso de "point-free" ser a simplicidade, vamos focar-nos na obtenção de expressões suficientemente simples para que possam ser manipuladas e verificadas.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Every program starts from an idea, a model, an abstraction, which is iteratively refined until we reach the final result, the implementation. However, at the end, one must ask: does the final program resemble in anyway the original model? Was the original idea correct to begin with? Formal methods guarantee that those questions are answered positively, resorting to mathematical techniques. In particular, in this thesis we are interested on the second factor: verifying the correctness of formal models.

Verifying models is obviously easier than verifying the final programs: models are much more abstract than programs, with less and simpler constructs. However, if the model is too abstract, it will not be able to correctly capture and abstract the implementation. As such, one must choose specifications that are neither complete abstractions, nor almost at the implementation level. One class of languages that fits that description is the one of *state-based modeling languages*. In these languages there is a notion of *state* that is defined by a set of variables and their types. This state may change upon the application of an operation, as long as the *pre-condition* for that operation holds before it is applied, and the *post-condition* holds after. There is also the notion of *invariant*, properties that must always hold. Also, these languages are structured, allowing the definition of complex datatypes. Even restricting the specification languages to that class, some are more oriented towards deriving implementations by model refinement, like VDM [BJ78] and B [Abr96], and thus are less abstract, while others focus more on the verification of the formal model itself, like Z [Spi89], whose models are heavily based on mathematic concepts.

The defenders of *lightweight formal methods* consider that the "classical" formal methods cannot efficiently achieve precisely what they were originally created for: writing specifications and analyzing a system [JW96]. In fact, most of those methods are so complex that the cost of creating the specification is too high to be integrated in the everyday software development process. When we move to the analysis and verification tasks, the cost becomes even more evident. The idea of the lightweight formal methods is precisely to reduce the complexity of the specification, and automate the analysis.

Alloy is precisely a lightweight formal specification language, created by Daniel Jackson and his colleagues at MIT. In Alloy, the models are described using a simple mathematical notation, a particular kind of relational logic. In fact, Alloy's lemma is that *everything is a relation* [Jac06], which results in a very simple notation. Its notation is a well-behaved subset of Z, more amenable to automated verification, and tool for model checking within a given scope is even provided. On the other hand, it also inherits some characteristics of object oriented languages, allowing the definition of a type hierarchy and navigational expressions.

However, sometimes model checking is not enough. In safety-critical systems, for instance, there is a need to make sure that the program is *always* correct, i.e, to perform *unbounded* verification. The only way to do this is to mathematically prove that the specifications are correct, which poses some new problems. The mathematical logic must be expressive enough to be able to represent the specifications, while still being sufficiently understandable. However, expressive logics are usually *undecidable*: it is not possible to define an automatic technique to check the validity of the

specifications. So, the best we can hope for is to perform manual proofs, eventually assisted by interactive theorem provers, like HOL [GM93] or Isabelle [Pau94], which perform semi-automatic proofs, guided by the user.

Since Alloy's basic elements are relations, the *relational logic* (RL), *first-order logic* (FOL) enhanced with relational operators, provides the natural framework to represent their specifications. However, we believe that using a *point-free* (PF) notation, a style where there are no variables or quantifications, as opposed to point-wise (PW), provides a simpler framework where proofs can be carried out by simple equational steps. This transformation is used as a kind of *Laplace transform*, where complex problems become simple. Quoting José Nuno Oliveira [Oli08]:

> "Theories "refactored" via the PF-transform become more general, more structured and simpler. Elegant expressions replace lengthy formulae and easy-to-follow calculations replace pointwise proofs with lots of "..." notation, case analyses and natural language explanations for "obvious" steps."

The *calculus of relations* (CR), a characterization of RL without variables, was first axiomatized by Alfred Tarski in 1941 [TG87]. However, CR is not as expressive as RL: it is equivalent to RL restricted to three quantified variables. It was not until Freyd and Ščedrov defined the notion of an *allegory* [Fv90] that the bases for the representation of programs in a relational setting were laid down. Allegories abstract CR in a categorical setting, allowing the definition of a categorical calculus of relations (CCR), which is as expressive as RL. Eventually, an algebra of programming based on the categorical CR was defined [BdM96, Bac04]. These techniques evolved from the PF equational calculus originated by John Backus [Bac78], which has already been widely and successfully used in program construction and verification [CPP06]. However, relations are better suited to represent some characteristics of programming languages, like partial functions and non-determinism, and they provide a higher degree of freedom in the calculus, since all relations have a well defined converse. The approach on PF RL we advocate is the one that has been developed by José Nuno Oliveira, with successful applications, for instance, in data refinement [Oli08, OCV06], extended static checking [Oli09] and operation refinement [OR06]. At the same time, another solution to the limited expressiveness of CR was being developed. *Fork algebras* (FA) [Fri02] are defined as an algebraic extension of CR, introducing a new operator, the *fork*. Likewise to CCR, FA have already been successfully used in program specification [BFHL96, FA94, FAN93, FPA04].

Our goal is to present a translation from Alloy specifications to a PF RL, which can then be mathematically proven, either resorting to proof assistants or to manual proving. A solution to this problem had already been proposed by Frias et al [FPA04], presenting a transformation from Alloy to FA. However, this approach has two issues: the resulting PF expressions are excessively complex, which goes against our motivation for using a PF notation, and do not consider the translation of the complete Alloy model, but only the translation of Alloy formulas.

We propose a new translation, this time from Alloy to CCR. Categories are typed, so they provide a better framework for representing the Alloy type system. Also, since our motivation for the use of PF is simplicity, we will focus on obtaining PF expressions that are simple enough for manipulation and conduct proofs about them. This translation resulted in an implementation of a tool, which converts Alloy specifications to the PF style.

We do not intend to replace the Alloy Analyzer in any way, but rather to work along side with it. The Analyzer is used to find all specification problems within a bounded environment. When all properties are verified by the Analyzer, and there is need for unbounded verification, we translate the specifications to PF, and try to prove them in that framework.

Overview We start by introducing the Alloy modeling language and its underlying logic in chapter 2. In chapter 3 we present the mathematical concepts needed to understand how we represent programs. We start with CR, and then present its two extensions, FA and CCR. In chapter 4 we present a RL to CCR translation. We start by analyzing the Alloy to FA translation from [FPA04], and then based on it, we present a simpler translation. In the end, we further simplify that translation by applying heuristic rules. In chapter 5 we present a transformation

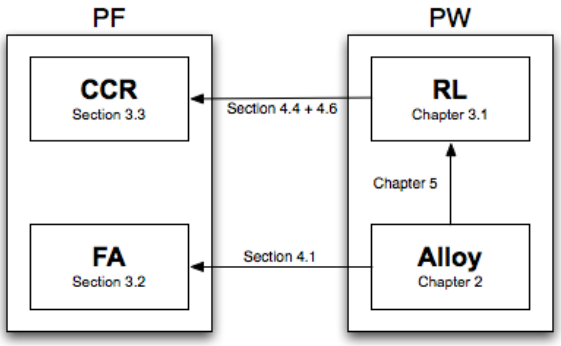


Figure 1.1: Overview of the formalisms and translations to be presented.

from Alloy specifications to RL. In chapter 6 we present how both translations, form Alloy to RL and RL to CCR were implemented. Finally, we present our conclusion and future work in chapter 7. Figure 1.1 resumes the formalisms presented and the translations between them.

Chapter 2

Alloy

Alloy [Jac02] was created by Daniel Jackson and his team at MIT and takes a different approach than the so called "classic" formal methods. It was built to be lightweight and, unlike B [Abr96], VDM [BJ78] or Z [Spi89], which focus on theorem proving, the emphasis is on automatic analysis. Defenders of *lightweight formal methods* [JW96] state that modeling languages should be easy to write, with a minimal, easily understood notation, and automatically verified, and Alloy fits exactly in that definition. It is based on Z [Spi89], but selects only the essential features, which results in a simpler language. Alloy's underlying logic is a kind of RL, making it easier to write and read specifications without the need to learn complicated logics. On the other hand, it is also influenced by object modeling languages, from which it inherits the navigational style and type hierarchy. Since it is a recent formal method, there are few tools available, but it benefits from the knowledge gathered by other previous methods. As for the verification of the model, a tool, the Alloy Analyzer is provided which automatically verifies the specifications, within a bounded scope.

2.1 Atoms and Relations

In Alloy *everything is a relation* [Jac06]. So, expressions and constraints are written in a kind of relational logic (see chapter 3) and, as such, there is not any typical constructor that we would find in a programming language. Therefore, Alloy is quite simple to learn: one only needs to know basic mathematical concepts, namely, set theory and FOL. However, these relations are more generic than binary relations: in Alloy, the arity of the relations is arbitrary.

The primitive entities of Alloy are *atoms*, which are indivisible, immutable and uninterpreted. This means that atoms cannot be integers, as they are interpreted[1], or even pairs, as they are divisible. So, to model anything remotely useful, we need to introduce relations between atoms.

Relations are just *sets of tuples*. If the relation is unary, containing tuples of arity 1, we simply call it a *set*. An uncommon characteristic in Alloy, is that scalars are also relations. In fact, a scalar is a set with only one element. Although this may seem unnatural, the uniformity that arises from the fact that everything is represented the same way really simplifies the definition of the properties, as well as the complexity of the tasks performed by the Analyzer.

2.2 Language

Briefly, an Alloy model consists of *signature* declarations, where atoms and relations are defined, *constraint* paragraphs, that define constraints and properties of the model, *assertions*, defining what we want to check, and *commands*, which instruct the Analyzer to perform particular analyzes.

[1] Alloy actually has some limited support for integers and operations on them, but this is an exception. They are presented in section 2.2.6.

Specific instances of the model, where a concrete set of atoms and relations between them are provided by the Analyzer to either show possible instances of the model, or counter-examples for an assertion.

We will now introduce the basics of the Alloy language and its underlying logic, based on the presentation in [Jac06] and using a running example.

2.2.1 An Example: a Memory model

Consider the model of a computer memory. For the sake of simplicity, we will consider that the memory consists of a set of *blocks* which are allocated between defined *bounds*. Each position of the block may or not have a *content*, which has a *value* and a *type*. Finally, we consider that there is a set which contains the bounds of the memory which are *free*. We first present a static version of this model in Alloy, which can be seen in Figure 2.1. In Section 2.2.8 we introduce the dynamic aspects, namely operations.

2.2.2 Signatures

The *signatures* (`sig`) introduce a set of atoms. For instance, the signature `Block` introduces a set which contains atoms of this type. This set can also be referenced in the constraints using the name of the signature: `Block` would represent the set of all atoms of type `Block` in the given instance. The number of atoms in the set in each instance is not fixed (it could even be empty). Alloy also supports *subtypes*. In the example, the concrete types of the blocks are defined as an *extension* of the *abstract* signature `Memtype`. All the extensions of a set are disjoint. Also, when a signature is labeled as abstract, there cannot be any atom in the model of that type: they must belong to one of the extensions. Extensions that are not necessarily disjoint could also be defined using the keyword `in`. If we would have used that keyword instead of `extends`, a atom of the set `Memtype` could represent more than one type.

Multiplicity operators may be used to limit the number of atoms in signatures or defining specific properties of the relations. For instance, when we declare a signature we have no guarantee concerning the cardinality of the respective set in an instance. However, we can use the keywords `set`, `one`, `some` or `lone` to force that set to contain any number, exactly one, at least one, or at most one atom, respectively. The default multiplicity is `set`. In our example, we use the multiplicity `one` to guarantee that there can only be one atom representing each memory type. Since `Memtype` is abstract, this means that this set is exactly `{(Int8),(Int16),(Int32),(Float32)}`. Note the use of parenthesis even in single elements, since they represented by unary relations.

2.2.3 Relations

Relations are introduced as fields of the signatures. For instance, in the signature `Content`, relation `type` states that the atoms of that signature will be related to the atoms of the signature `Memtype` by a relation named `type`. These can also be n-ary relations, as we can see with the relation `contents` in the signature `Memory`, which relates memories, blocks, integers and contents.

Relations can also be restricted by the multiplicity operators already presented. For instance, in our example, in the signature `Content`, the relation `contents` relates each atom `Content` to *exactly* one atom `Value`. We introduced two fields with the same name on purpose, to explore the Alloy overloading system. Considering now a n-ary relation, in the signature `memory`, the relation `bounds` relates memories, blocks *at most one* bound. The result is a ternary relation where, for a particular memory, any `Block` contained in it is related to only one `Bound`, but not all blocks must belong to it.

Properties over the relations may also be defined in the signature, after the relation declarations, like in the restriction of the `size` of the memory types in our example. Note that in this case, relation occurrences are assumed to be universally quantified on the domain. These properties are known as signature facts.

```
module Memory
open util/integer

sig Memory{
  frees : set Bound,
  bounds : Block → lone Bound,
  contents : Block → Int → lone Content
}
sig Block{}
sig Bound{
  high : one Int,
  low : one Int − high
}
sig Content{
  type : one Memtype,
  contents : one Value
}
abstract sig Memtype{
  size : Int
}
one sig Int8 extends Memtype{}{size = 1}
one sig Int16 extends Memtype{}{size = 2}
one sig Int32, Float32 extends Memtype{}{size = 4}
sig Value{}

fun sizeof[m : Memory, b : Block] : Int{
  sub[b·(m·bounds)·high,b·(m·bounds)·low]
}

pred valid_access[m : Memory, t : Memtype, b : Block, i : Int]{
  lte[b·(m·bounds)·low,i]
  lte[add[i,t·size],b·(m·bounds)·high]
}

fact{
  all m : Memory, b : Block |
    b·(m·bounds)·low < b·(m·bounds)·high
}

assert validtypes{
  all m: Memory, b: Block, i: Int |
    valid_access[m,i·(b·(m·contents))·type,b,i]
}

run valid_access for 5

check validtypes for 1 Memory, 4 Block, 3 Bound, 3 Content, 2 Value
```

Figure 2.1: An example of a Memory model in Alloy.

Relations may also be defined as *dependent* on other relations, as long as they are defined in the same signature, or at its supertype. For instance, consider the definition of the relation `low` in the signature `Bound`. It depends on the relation `high`, which in this case means that a tuple must not be contained in both (a block must not start and end at the same position). We can also see that relations defined in the supertype may be restricted by the subtypes. The atoms `Memtype` are related to an integer which represents its size, but each subtype restricts that integer to a concrete one, using signature facts like `size = n`, for some integer n.

2.2.4 Constraints

Constraints in Alloy are used to restrict the model, or to check if certain property holds true. They may be *facts*, *predicates*, *functions* or *assertions*.

Predicates and functions are used to create reusable expressions, which can be used in other constraints. Functions (`fun`) are expressions that relate a fixed number of typed arguments, and returns another expression. In our example, the function `sizeof` receives a block as an argument and return its size. We see for the first time the use of the relational operator *dot join* (`.`), which represents the relational composition presented in section 3.1.

Predicates (`pred`) are constraints which relate two expressions passed as arguments. The predicate `valid_access` tests if in a given memory, a given content type and a given position, is within the bounds of a given block. The predicate succeeds if the expressions contained in its paragraph both succeed: the position is equal or higher than the lower bound of the block, and the position plus the space occupied by the given content type is equal or lower than the higher bound of the block. `lte` and `add` are also predicates, and are implemented in the integer library of Alloy, which will be explained in a later section. Note that the predicate asks for a memory, instead of just using the keyword `Memory` to retrieve the memories of the system. We will soon show why this kind of expressions are so important.

The previous constraints do not force or test if properties are valid in a model. For that purpose we need facts and assertions, which may use the defined predicates and functions. Facts (`fact`) are used to describe properties that must always be true, that is, assumptions of the model. They must be used to state constraints that we know that will always hold in the implementation of the system: facts are not used to check if properties are true, they simply are. In our model, we introduced a fact to state that the lower bound of a block is always lower than the higher bound. This may or may not have the intended effect. If we intended to check if the operations always created blocks with valid bounds, we will no longer be able to check it, since that fact already forces them to be so.

Finally, *assertions* (`assert`) are used to check if an expression holds in a model. Unlike facts, assertions search the space of possible instances for counter-examples, instances where the property does not hold. This is very useful, since, it not only finds possible bugs on the model, but also presents the instance in which they appeared. In our example, the assertion `validtypes` checks if in each position of the block, the type of the content if valid, concerning the bounds of the block. This may or not be true, depending on the other constraints of the model. In our case, there is nothing that would make it hold, so the Analyzer would find a counter-example. We can also see here why it was important to pass the memory as an argument of the predicate: it now allows us to check its validity for *all* memories, as it is universally quantified in the assertion.

2.2.5 Operators and Constants

We have shown how to create the paragraphs which constrain the model. We will now present how the properties are actually expressed. Alloy's operators are based on the ones from RL. This allows us to express the properties in a variety of ways.

Concerning the relational calculus, the operators are divided in two groups, which arise from the fact that the relations can be seen as actual relations or as sets of tuples: *relational operators* and *set operators*. The set operators are the classical ones: *union* (`+`), *intersection* (`&`), *difference* (`-`), *subset* (`in`) and *equality* (`=`), which, since variables are also relations, may either represent

variable equality or relation equality. These can be applied to sets of different types of atoms, as long as they have the same arity. The relational operators require a more detailed explanation.

The *dot join* (`.`) represents the relational composition, and is one of the most important operators. However, it can be applied to relations that are not binary, unlike the one from relation algebra (see section 3.1). Generically, it joins two relations matching the last element of the first with the first of the second, discarding that element. One particular use is when one of the relations is unary, i.e. a set. In that case, composition works like an application of the other relation to the elements of that set[2]. This method of accessing the elements of a relation is called *navigation*. Lets take a closer look at the expression `b.(m.bounds)`, that appears multiple times in our example. The relation `bounds` has the type `Memory -> Block -> Bounds`. By applying a memory `m`, which is a set, since scalars are represented by a set with only one element, to that relation, we get a relation of the type `Block -> Bounds`, i.e., the bounds of all blocks contained in the memory `m`. If we then apply another set to that relation, this time a block, we will get the bounds of that block. The *box join* (`[]`) is semantically identically to the dot join, but the arguments are given in a different order, i.e., `b.(m.bounds)` is equivalent to `m.bounds[b]`.

The *product* (`->`) operator is used to create relations (or tuples), using the cartesian product. For example, the command `Block -> Bound` would create a binary relation containing all the combinations of blocks with bounds (remember that `Block` and `Bound` represent the sets of all atoms of those types existing in each instance).

The *converse* (`~`) operator is equivalent to the converse ($^\circ$) operator from relation algebras, which reverts the elements of each pair. Note that it can only be applied to binary relations. The *transitive closure* (`^`) and the *reflexive-transitive closure* (`*`) are equivalent to the relational operators with the same name. Finally, the *domain* and *range restrictions* (`<:` and `:>`) select the elements of a relation whose domain (or range) is contained in a given set. For instance, if we had a set `b` of blocks, `b <: m.bounds` would create a subset of the binary relation `m.bounds` in which the first elements of the tuple are contained in `b`. Note that in Alloy, the first elements of the tuples constitute the domain, and the last the range. Also, the notion of domain in Alloy is different than the mathematical one, since it does not represent the set of all *possible* values of a relation, but the set of the values actually contained in the relation.

Besides these operators, Alloy also provides some constants. The first, `univ`, represents the *unary universe* of the model. The `iden` represents the *binary identity* relation. Note that this relation is not typed, since it contains the identity tuples for all atoms. The `empty` represents the empty set.

As for FOL, the operators available are the common ones: *not* (`!`), *and* (`&&`), *or* (`||`), *implies* (`=>`) and *iff* (`<=>`).

Alloy supports some quantifiers that are not defined in standard presentations FOL. Asides from the common *forall* and *exists*, respectively `all` and `some` in Alloy, it also accepts `no`, `one`, and `lone`, meaning none, only one, or at most one element satisfies the property, respectively. It also supports nested quantification and multiple quantification. Alloy also supports higher-order quantification, thus is not restricted to FOL. However, these constraints cannot always be analyzed. Quantifiers may also be applied directly to expression, in which case they test their cardinality.

Finally, Alloy also supports the definition of relations by *comprehension* and allows the use of some auxiliary constructs to help build the constraints, namely *let* and *if-then-else* expressions.

2.2.6 Integers

Although atoms are uninterpreted, an Alloy library introduces integers as a special class of atoms. Each *integer value*, type `int`, has an associated *integer atom*, `Int`. This is similar to Java's integer types `int` and `Integer`, respectively. When handling a numeric expression `e`, `Int e` denotes the set of integer atoms holding the result of `e`, and `int e` denotes the sum of the integer values of those atoms.

[2]The dot join of two sets is invalid, since there are no 0-ary relations in Alloy.

The number of integer atoms in an instance, represented by the set `Int` is limited, and defined by a scope, explained in 2.5. This greatly limits the use of integers in Alloy, as, for instance, integer overflows often occur. So, before using integers, one must consider if they are really necessary. If we only need them as an identifier, any generic atom would do. If we only need a total order, the library `ordering` would probably solve the problem. Only when arithmetic constraints are necessary is the use of the integer atoms mandatory.

Considering our example, we needed to use integers because we want to check, for instance, the overlapping of block bounds. However, most constraints presented will generate false counterexamples, as we have omitted the expressions which check for integer overflows.

2.2.7 Commands and Scope

The verification of the model is performed by the Analyzer, which will be explained later. However, one must write a *command* which will instruct the Analyzer on how to behave. There are two possible approaches. With *run*, the Analyzer tries to find an instance that satisfies all the facts plus the given predicate. For instance, the run command in our example finds an instance of the model in which all blocks would be valid. The command *check* instructs the Analyzer to check if an assertion is true. This time, the Analyzer tries to find an instance in which the assertion does not hold, a counter-example. In our example, the check command would try to find an instance of the model in which the allocated blocks would have invalid types.

The scope of the analysis can also be specified in these commands. One can set the scope of all signatures, like we did in our `run` command, restricting it to 5 atoms per signature, or set the scope individually for each signature, like in our `check` command.

2.2.8 Building up the example: dynamic operations

The version of the memory model we presented was static. It allows us to analyze instances of the model, but not its evolution. We cannot, for instance, verify that, beginning with a empty memory, and applying to it only valid operations, the state memory never transits to an invalid one.

Lets now write some operations. In Alloy, these can be specified as properties in the signatures or using predicates that relate two states. Lets return to our example, and define an operation on the memory, say *alloc*. We need to define a predicate that relates the states before and after the allocation. A possible definition is presented in Figure 2.2.

Writing operations as logical predicates has some particular characteristics. First, we see that the predicate receives the states of the memory as arguments. In our case, `m` represents the pre-state of the memory, and `m'` the post-state. The block to alloc is `b` and `s` is the space it will occupy. Note that a predicate is a conjunction of all the lines contained in the paragraph. This means that it will only be applied if all the expressions hold. So, one can write pre-conditions of the operations inside the predicate: the operation will only be applied if the pre-condition is true. These are usually expressions that apply only to arguments other than the post-state. In our example, `pos[s]` is a pre-condition that states that the size of the block must be positive. Since we are relating states, not only do we need to state what changes, but also what stays equal, the so called *frame conditions*. In our example, we see in the seconds, third and fourth expression, that every tuple of the relations that does not involve the block `b`, is not changed by the operation. Then, in the next two lines, we simply say that in the post-state the new block must not be free or have contents. This means that if that block already had contents, they will disappear in the new state. However, this also means that the it will not have any random content in the new state. Remember that, since this is a predicate, it is important to restrict every possible non desired state.

Now that we have stated what is not changed in the memory, we will move to the actual modifications, given by the last three lines. Note that we do not tell the Analyzer how to create the new state: we only say that that the lower bound must be positive, the difference between the higher and lower bound must be equal to the desired size, and that the new block must not

```
pred alloc [m : Memory, s : Int, b : Block, m' : Memory]{
  pos[s]
  m'·bounds − (b → Bound) = m·bounds − (b → Bound)
  m'·frees − b = m·frees − b
  m'·contents − (b → Int → Content) = m·contents − (b → Int → Content)
  b not in m'·frees
  b not in m'·contents·Content·Int
  pos[b·(m'·bounds)·lower]
  b·(m'·bounds)·higher = add[b·(m'·bounds)·lower,s]
  all blck : (m'·bounds·Bound − b) | not overlap[m',blck,b]
}

assert allocOK{
  all m, m' : Memory, b : Block, i : Int |
    inv[m] && alloc[m,i,b,m'] ⇒ inv[m']
}

check allocOK for 3 but 2 Memory
```

Figure 2.2: An example of an operation modeled in Alloy.

overlap any of the existing. Other than that, the Analyzer is entirely free to create the new state, meaning that it is highly non-deterministic.

To verify that this operation is correct, we can simply check if the invariant holds: applying the operation to a valid memory, the memory stays valid. Assuming that there exists a predicate `inv` that contains all the needed constrains of the memory, one can check the correctness of the operation using the assertion `allocOK` written in the example.

2.3 Notation styles

As we can deduce from the high level of abstraction of Alloy, expressions can be written in a variety of styles. More precisely, Alloy supports three different styles, which can be mixed at will [Jac06]. Each one is closely related to a different logic.

We can write expressions using FOL, with quantifiers and boolean expressions. Lets see a typical example, stating that a relation $R : A \to B$ is injective. In FOL this could be defined as:

```
all a, a' : A| some b : B | b = a.R && b = a'.R implies a = a'
```

In the *navigation expression* style, the dot join is used to navigate through the relations. It also allows quantifiers.

```
all b : B |  lone R.b
```

The *PF style* consists of expressions including only relational operators, with no quantifiers.[3]

```
R.~R in iden
```

This style is obviously simpler than the other two, albeit less comprehensible. It is, in fact, the point-free notation we are looking for. So our main purpose in this thesis is to research how to reduce the two other styles to this one.

[3]See the presentation of the taxonomy of relations in section 3.3.2 on why this is true.

2.4 Type-checking system

In Alloy, the type checking system has two functions: detecting type errors, and resolving overloading of fields with the same name. However, the notion of type error in Alloy is not a common one. An expression is incorrect if it is proven to be *irrelevant*.

Like everything in Alloy, the types are also treated as relations. In fact, each type is an over-approximation of the expression it represents. The detailed presentation of the type system can be found in [EJT04]. We will just briefly explain it using a concrete example.

Each signature introduces a *basic type* into the system. Each extended signature is a *subtype*, and subsets acquire the type of its parent. Using our running example, we would have the types `Memory`, `Block`, `Bound`, `Content`, `Int`, `Memtype`, `Value`, `Int8`, `Int16`, `Int32` and `Float32`. `Int8`, `Int16`, `Int32` and `Float32` are subtypes of `Memtype`.

To each expression corresponds a *relational type*, which is a union of products. For instance, the relation `frees` would have the type `Memory -> Bound` and the relations `bounds` would have the type `Memory -> Block -> Bound`. The type resolution of most expressions uses the same rules for the types as it would for the relations. For instance, `frees + type` would have the type `(Memory -> Bound) + (Content -> Memtype)`, and the type of `contents.type` would be `Memory -> Block -> Int -> Memtype`. Using this method, the *bounding type* of each expression is determined. Regarding our example, lets consider the expression:

```
Content.contents
```

Note that there are two fields named `contents`. To resolve the overload, Alloy distinguishes both, and considers the union:

```
Content.(contents_M + contents_C)
```

where `contents_C` represents the relation from signature `Content`, and `contents_M` the one from signature `Memory`. Expanding the relational types, and applying the joint operator, the resulting bounding type of this expression would be `Value`.

Once the bounding types are calculated, they are used to find the irrelevances in the expression and resolving the overloadings. This analysis is done top-down. So, to find the relevance type of `(Block + Content).(contents_M + contents_C)`, we check which types contributed to the final type, `Value`. In `contents_M + contents_C` we see that only `contents_C` contributes to the type. Thus, the overloading of the `content` fields is resolved.

If, for instance, the expression was `Block.(contents_M + contents_C)`, its bounding type would be empty, and the Analyzer would sent an error. Actually, in practice, these errors are actually seen as warning by the Analyzer, since it can still perform the verification. The only type mismatches that the Analyzer does not support are the one involving relations of different arities.

2.5 The Analyzer

As we have seen, Alloy's models can be analyzed using the commands `check` and `run`. Both produce *instances* of the model: checking tries to produce a counter-example instance in which the constraints are not satisfied, and running generates one in which they are. So, although they represent different approaches, they all reduce to the same problem.

In spite of the name, Alloy Analyzer does not really analyze the model. It translates the specifications into boolean formulae, which then are interpreted by any off-the-shelf SAT analyzer. Every relation is translated to a matrix of boolean values, in which each column represents the possible values in the relation. Relational expressions are then translated to boolean expressions. For instance, the union of two sets is represented by the disjunction of the respective matrices.

Alloy's logic is undecidable but, by the principles of lightweight formal methods, the analysis of the model should be automatic. So, compromises had to be made: the notion of *scope* was introduced. The scope defines how many elements each signature will have, and it can have different values for each type. For instance, in our example, we run the model with only one

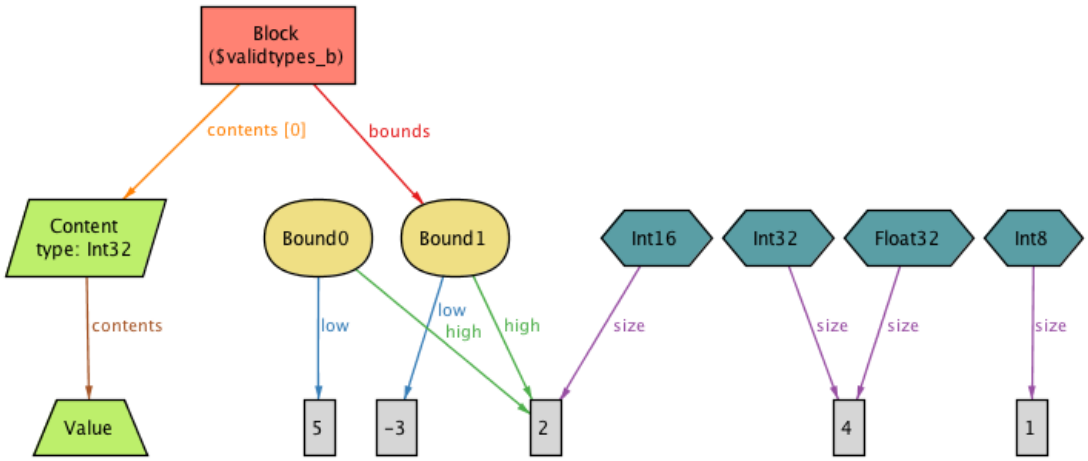


Figure 2.3: An example of an Alloy snapshot.

`Memory`, 4 `Block` and 3 `Bound`. These numbers may seem too small, and not representative of the reality. However, note that, for instance, a binary relation relating 3 possible atoms has 512 possible combinations. Consider higher arity relations and the state explosion is evident. Also, the creators of Alloy also believe in the "small scope hypothesis": most bugs have small counterexamples [Jac06]. So, any bug found in large instances will most likely also be found in small ones.

Despite that, the existence of a scope makes a class of properties impossible to prove. These are usually properties which assume that the signature set contains all possible elements. This happens, for instance, in constraints of the type

```
all s1,s2 : Set | some s3 : Set |
  s3 = s1 + s2
```

In this case, there is no guarantee that the set `s3` will exist. For a more detailed explanation of this problem see [Jac06]. Although this is, in theory, a valid constraint, in a finite universe it is not.

Other problems with the limited scope arise with the use of integers. When defining scope, the keyword `int` defines the maximum bit-width of the integer values, i.e., the range of the values. The fact that their number is limited makes integer overflow a common problem with arithmetic constraints. Although there are ways to avoid these, Alloy is not a natural language to deal with integers.

The Analyzer presents the instances graphically, which may be manipulated to improve readability. For instance, since there is no intuitive way to graphically present n-ary relation, one can project out some values being presented. Figure 2.3 presents a snapshot of our model obtained using the command `check validtypes`, which returns a counter-example. Note that to improve readability, `Memory` is projected.

Chapter 3

PF Relational Logic

Although functional calculus is usually used in program specification, the generalization from functions to relations provides many benefits. Most so called functions in functional programming are partial, so they are actually better represented by relations. Also, all relations have a well defined conversed, which makes formula manipulation easier. Finally, since relations are essentially non-deterministic they provide a natural way to represent non-deterministic problems. If we then consider a point-free characterization of RL, to those advantages we add those of the point-free reasoning: simplicity and ease of manipulation.

Relational logic (RL), first-order logic (FOL) enriched with some relational operators. The point-free characterization of RL is called the *calculus of relations* (CR). Standard CR however, is not as expressive as RL. To overcome that limitation, different approaches were created, two of which we will now present.

3.1 Calculus of relations

CR will now be briefly presented. We intentionally oversimplified some concepts, so, for a more in-depth presentation, see [Mad06].

The study of CR began with the works of Augustus De Morgan [Mor60] and Charles Sanders Peirce [Pei33] in the 19th century, and later with Ernst Schröder [Sch95]. From their works the widely known algebra of binary relations was eventually derived.

Definition 3.1. The structure $\langle U, \cup, \cap, ^-, \bot, \top, \cdot, id, ^\circ \rangle$ is called a *algebra of binary relations* (ABR) (or proper relation algebra), where the universe U is a set of binary relations on a set A and satisfies:

1. $\top$ is largest relation, i.e., $\bigcup R \in U \subseteq \top$, id is the identity relation on the set A and $\bot$ is the empty relation,

2. $\cup$ is the union of relation, $\cap$ the intersection of relations and $^-$ the complement of relations relative to $\top$,

3. $id, \bot, \top \in U$

4. U is closed under $\cup$, $\cap$ and $^-$,

5. U is closed under relational composition ($\cdot$) and converse ($^\circ$), which are defined, for all $R, S \in U$, by:

$$R \cdot S = \{\langle a, b\rangle : \langle \exists\, c : a\, R\, c \wedge c\, S\, b\rangle\}$$

$$R^\circ = \{\langle a, b\rangle : b\, R\, a\}$$

Although these authors were interested in reasoning about complicated formulas using relations, they did not intend to axiomatize the calculus. In 1941, Alfred Tarski introduced the elementary theory of binary relations (ETBR) [Tar41] as a logical formalization of the algebras of binary relations. In ETBR two sets of variables are considered: the *individual variables* and the *relation variables*. If we add to the set of the relation variables the *relation constants* 0 (absolute zero), 1 (absolute unit) and $1'$ (identity), and close the set under the binary operators + (absolute addition), . (absolute product) and ; (relative product) and the unary operators $^-$ (complement) and $^\smile$ (converse), we obtain the so-called *relation designations*. Atomic formulas are then of type $x\,R\,y$ or $R = S$, with x and y individual variables and R and S relation designations. From atomic formulas we obtain the compound formulas from the typical logical operators $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$, and the quantifiers $\forall$ and $\exists$ over individual variables.

ETBR is as formalization of FOL, which we nowadays usually refer to as RL. From there, Tarski [Tar41] defined CR as a subset of RL with no individual variables, i.e., atomic formulas are of the form $R = S$.

Relation algebras, are the result of Tarski's axiomatization of CR:

Definition 3.2. A *relation algebra* (RA) (or abstract relation algebra) is an algebra $\langle A, +, \cdot, ^-, 0, 1, ;\ , 1', ^\smile\rangle$ where + , $\cdot$ and ; are binary operations, $^-$ and $^\smile$ are unary, and 0, 1 and $1'$ are distinguished elements. Furthermore, the reduct $\langle A, +, \cdot, ^-, 0, 1, ;\ \rangle$ is a boolean algebra and the following identities are satisfied for all $x, y, z \in A$:

$$x\,;(y\,;z) = (x\,;y)\,;z \tag{3.1}$$

$$(x + y)\,;z = x\,;z + y\,;z \tag{3.2}$$

$$(x + y)^\smile = x^\smile + y^\smile \tag{3.3}$$

$$(x^\smile)^\smile = x \tag{3.4}$$

$$x\,;1' = 1'\,;x = x \tag{3.5}$$

$$(x\,;y)^\smile = y^\smile;x^\smile \tag{3.6}$$

$$x^\smile;y \cdot z = 0 \text{ iff } z\,;y^\smile \cdot x = 0 \text{ iff } x^\smile;z \cdot y = 0 \tag{3.7}$$

These axioms define an abstract algebraic structure, for which a concrete implementation are the ABR already presented, when we take the absolute addition + as the union of relations $\cup$, the absolute product $\cdot$ as the intersection of relations $\cap$ and the relative product ; as the composition of relations $\cdot$.

At the end of the article, Tarski presented some questions about the axiomatization. One that particularly interests us is if every formula of RL is expressible in CR. The answer to this question is due to Korselt [L̈15] and had been known for several years. In fact, the CR is equivalent to a three-variable restriction of FOL [TG87]. For instance, the following formula, from [TG87], is not expressible in CR:

$$\langle \forall\, x, y, z : \langle \exists\, u : u\,\overline{1'}\,x \wedge u\,\overline{1'}\,y \wedge u\,\overline{1'}\,z\rangle\rangle \tag{3.8}$$

The relation between RL and CR is the same as the one between our PW and PF relational notation. In fact, thePF transformation is largely based on Tarski's works, and its base is a RL to CR translation. However, this limitation of CR poses a serious limitation on our expressiveness power. Fork algebras are one of the mechanisms developed to overcome this limitation.

3.2 Fork Algebras

As has been presented, the CR is equivalent to a three variables fragment of RL. Since our goal is to represent program specifications, that restriction highly limits the expressiveness power. Fork Algebras were developed to overcome that limitation, with the goal of program specification,

validation and derivation, for which they have already been successfully used [BFHL96, FA94, FAN93, FPA04, HBS93, HBS93, HV91].

Briefly, fork algebras extend ABR with an operator ∇ (called fork) and a binary function $\star$, where:

$$y\,R\,x \wedge z\,S\,x \Leftrightarrow \star(y,z)\,R\nabla S\,x \tag{3.9}$$

The study of these algebras began with [HV91], where fork algebras are built using finite trees, $\star(x,y)$ representing the tree with x and y as sub-trees. Although other bases for fork algebras were proposed, like in [VH91], where they are represented as finite strings, it was proved in [MSS92] that none of those are finitely axiomatizable. Although the current definition of fork algebras [Fri02], with $\star(x,y)$ denoting the application of a binary injective mapping to x and y, was presented in [VH93], it contained an extra non-equational axiom to achieve representability, so the current axiomatization was first presented in [HBS93].

Using fork algebras we can now express formula 3.8 without variables:

$$\begin{aligned}
&\langle \forall\, x,y,z : \langle \exists\, u : u\,\overline{1'}\,x \wedge u\,\overline{1'}\,y \wedge u\,\overline{1'}\,z)\rangle\rangle \\
\Leftrightarrow\ &\langle \forall\, x,y,z : \langle \exists\, u : u\,\overline{1'}\,x \wedge u\,\overline{1'}\nabla\overline{1'} \star (y,z))\rangle\rangle \\
\Leftrightarrow\ &\langle \forall\, x,y,z : x\overline{1'}^{\smile};\overline{1'}\nabla\overline{1'} \star (y,z)\rangle \\
\Leftrightarrow\ &\langle \forall\, x,y,z : x\overline{1'}^{\smile};\overline{1'}\nabla\overline{1'} \star (y,z) \Leftrightarrow \mathit{true}\rangle \\
\Leftrightarrow\ &\langle \forall\, x,y,z : x\overline{1'}^{\smile};\overline{1'}\nabla\overline{1'} \star (y,z) \Leftrightarrow x1y \wedge x1z\rangle \\
\Leftrightarrow\ &\langle \forall\, x,y,z : x\overline{1'}^{\smile};\overline{1'}\nabla\overline{1'} \star (y,z) \Leftrightarrow x1\nabla 1 \star (y,z)\rangle \\
\Leftrightarrow\ &\overline{1'}^{\smile};\overline{1'}\nabla\overline{1'} = 1\nabla 1
\end{aligned}$$

3.2.1 Definition

The definition of fork algebras we use is the one presented in [Fri02]. The class of *proper fork algebras* (PFA) is an extension of ABR [Tar41] with fork, which induces a structure on the base of the algebras. The domain of the binary relations is no longer a plain set, but a structured domain $\langle A,\star\rangle$, where $\star$ is an injective binary function on A, i.e.,

$$\langle \forall\, x,y,u,v : \star(x,y) = \star(u,v) \Rightarrow x = u \wedge y = v\rangle \tag{3.10}$$

Definition 3.3. A *proper fork algebra* is a two-sorted algebraic structure $\langle U,\cup,\cap,^{-},\bot,\top,\cdot,id,^{\circ},\nabla\rangle$ where the universe U is a set of binary relations on a structured domain $\langle A,\star\rangle$, such that:

1. $\langle U,\cup,\cap,^{-},\bot,\top,\cdot,id,^{\circ}\rangle$ is an algebra of binary relations on a set A,
2. U is closed under fork of binary relations, defined by:

$$R\nabla S = \{\langle x,\star(y,z)\rangle : x\,R\,y \wedge x\,S\,z\} \tag{3.11}$$

where $\star : A\times A \to A$ is a binary function that is injective on the restriction of its domain to $\top$.

Since $\top$ is a binary relation on A, the restriction of the injectivity of $\star$ to $A \to A$ is adequate. The function $\star$ performs the role of pairing, encoding pairs of objects into single objects. It is

important to note it is distinct from the set-theoretical pair formation, i.e., there are models where $\star(x, y)$ is not the same as (x, y).

Given a pair of relations, the operation *cross* (denoted by $\otimes$) performs the relations in parallel, defined by:

$$R \otimes S = \{\langle \star(x, y), \star(w, z)\rangle : x\ R\ w \wedge y\ S\ z\} \tag{3.12}$$

Much like RA are an abstract counter-part of ABR, proper fork algebras also have an abstract version: abstract fork algebras (AFA). This class is also finitely axiomatized with a set of equations.

Definition 3.4. An *abstract fork algebra* is an algebraic structure $\langle R, +, \cdot, ^-, 0, ;\ , 1', ^\smile, \nabla\rangle$ where $\langle R, +, \cdot, ^-, 0, ;\ , 1', ^\smile\rangle$ is a relation algebra and for all $r, s, t, q \in R$,

$$r\nabla s = (r\,;(1'\nabla 1)) \cdot (s\,;(1\nabla 1')) \tag{3.13}$$

$$(r\nabla s)\,;(t\nabla q)^\smile = (r\,;t^\smile) \cdot (s\,;q^\smile) \tag{3.14}$$

$$(1'\nabla 1)^\smile \nabla (1\nabla 1')^\smile \leq 1' \tag{3.15}$$

When in a proper fork algebra, the operations $(1'\nabla 1)^\smile$and $(1\nabla 1')^\smile$are quasi-projections [TG87], and will be represented by π and ρ, respectively.

Considering the Def. 3.4 and the definition of the quasi-projections, the cross operation can now be defined by:

$$R \otimes S = (\pi\,;R)\nabla(\rho\,;S) \tag{3.16}$$

The simple equational rules of AFA provide an easy calculus, understandable even for the non-experts. However, in order to write specifications an easily understandable semantic is needed, which is a problem for AFA, since it is not clear how they look like. On the other hand, PFA, whose objects are binary relations, are a better candidate, as its operators are widely known and easily understood. So, the need arises to relate AFA with PFA.

This is known as the *representability* problem for FA. Since AFA are defined by a set of equations, the class is closed under isomorphisms, so the best we can hope is for AFA to be isomorphic of proper ones. Although we abstain ourselves from showing the proof, it was obtained independently by Gyuris [Gyu95] and Frias et al [FHV97], based on results by Tarski about the representability of quasi-projective algebras [TG87]. This result allows us to use the well defined objects of PFA, binary relations, based on the axioms of AFA, which simplify the reasoning of FA expressions.

3.3 Categorical Calculus of Relations

The category theory [Fv90] provides a way to abstractly reason about mathematical structures, and the relationships between them. The central idea is to generalize any mathematical constructions as objects and arrows between them. Categories and its central concepts were first introduced by Samuel Eilenberg and Saunders Mac Lane in 1942–45 [EM45], but it was the definition of allegories by Freyd and Ščedrov [Fv90] that allowed the CR to be defined in a categorical framework, which we will denote by *categorical calculus of relations* (CCR).

Categories have been widely used in program construction, since it provides a framework for datatype specification, while keeping those datatypes abstract. Our approach is based on the work of Bird and Moor [BdM96], which CCR to reason about programs.

3.3.1 Functions and Categories

Definition A *category* **C** is an algebraic structure consisting of a collection of *objects* (A,B,C,...) and a collection of *arrows* (f,g,h,...), as well as four operations. The first two operations are total and assign a *source* and a *target* objects to an arrow f, which is denoted by $f : A \leftarrow B$ (pronounced

'f is of type A from B'). Every object is a also connected to itself by an *identity arrow*, i.e., for an object A there is an arrow $id_A : A \leftarrow A$.

Finally, there is a partial operation called *composition* which takes two arrows f and g and creates a new one, $f \cdot g$. It is defined if and only if $f : A \leftarrow B$ and $g : B \leftarrow C$, in which case $f \cdot g : A \leftarrow C$ (pronounced 'f after g'). Composition is associative and has identity arrows as units, i.e., for all $f : A \leftarrow B$, $g : B \leftarrow C$ and $h : C \leftarrow D$:

$$f \cdot (g \cdot h) = (f \cdot g) \cdot h \tag{3.17}$$

$$id_A \cdot f = f = f \cdot id_B \tag{3.18}$$

Whenever the type of the identity arrow is irrelevant or easily deduced, we simply denote it by id.

Note that the order of the domain and range of the categorical arrows is different than the one from the previously presented logics.

Concrete categories A concrete example of a category is **Fun**, the category of sets and total functions. In **Fun**, objects are sets and arrows are typed total functions (f, A, B), where A is the range and B is the domain of f. The identity arrows are identity functions and the composition of arrows is the typical composition of functions.

A generalization of **Fun** is the category **Rel**, the category of sets and relations. An arrow $R : A \leftarrow B$ is a subset of the cartesian product $A \times B$. The identity arrow $id_A : A \leftarrow A$ is defined by:

$$id_A = \{(a, a) \mid a \in A\} \tag{3.19}$$

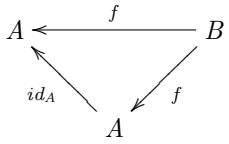
and the composition of (R, A, B) with (S, B, C) is the arrow (T, A, C) where

$$a \; T \; c = \langle \exists \, b : a \; R \; b \wedge b \; S \; c \rangle \tag{3.20}$$

Since total functions are particular cases of relations, **Fun** is a *subcategory* of **Rel**.

Note that order of the input and output of a relation are interchanged in relation to the relation algebras. This actually makes sense if we see relations as an extension of functions, which also take the input on the right-hand side.

Diagrams Since everything is represented by arrows, *diagrams* provide an intuitive way to represent information. In a diagram, $f : A \leftarrow B$ is represented by $A \xleftarrow{f} B$. For example, the equation $id_A \cdot f = f$ could be represented as:



When different paths between a pair of objects represent the same arrow, like in this case, the diagram is said to *commute*.

Duality We define for any category $\mathbf{C}$ the *opposite* category $\mathbf{C}^{op}$, which has the same objects and arrows of $\mathbf{C}$, but their source and target are interchanged. Also, composition is defined by swapping the arguments. Since reversing arrows twice does nothing, $(\mathbf{C}^{op})^{op} = \mathbf{C}$. Categories benefit from a special case of symmetry called *duality*, which means that any statement valid in a category is valid in its opposite.

Another interesting result is that $\mathbf{Rel} = \mathbf{Rel}^{op}$, because every relation, by definition, has a converse, and thus every arrow in **Rel** also has a converse.

Isomorphisms An *isomorphism* is an arrow $i : A \leftarrow B$ such that there exists an arrow $j : B \leftarrow A$ for which $j \cdot i = id_B$ and $i \cdot j = id_A$. The arrow j is unique, being called the *inverse* of i, and denoted by i^{-1}. In the **Fun**, these arrows represent bijections.

Terminal and Initial objects A *terminal* object T of a category $\mathbf{C}$ is an object such that, for any object A from $\mathbf{C}$, there is exactly one arrow $T \leftarrow A$. All terminal objects are isomorphic, so we will refer only to *the* terminal object, which will be denoted by 1, and the unique arrow $1 \leftarrow A$ by $!_A$.

The uniqueness of arrows is usually defined by *universal properties*. In the case of $!_A$, it is defined by

$$h = !_A \equiv h : 1 \leftarrow A \tag{3.21}$$

Terminal objects represent a datatype with only one element. In **Fun** it is a singleton set, which means all singleton sets are isomorphic in some way, and the arrow $!_A$ maps every element of A to the element of the singleton set. In **Rel** it is the empty set, and the arrow is the empty relation .

An *initial* object I of $\mathbf{C}$ is an object for each there exists exactly one arrow $A \leftarrow I$, for all objects A of $\mathbf{C}$. An initial object of $\mathbf{C}$ is a terminal object in $\mathbf{C}^{op}$. All initial objects are also isomorphic, so we refer only to *the* initial object, and denote it by 0. The unique arrow $A \leftarrow 0$ is represented by $¡_A$.

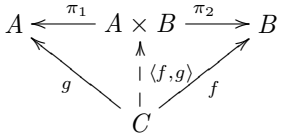
In **Fun** the initial object is the empty set, and the arrow $¡_A$ is the empty function. Note that the names 1 and 0 represent the cardinality of its sets in **Fun**. In **Rel** the initial object is also the empty set.

Functional product and coproduct We will now briefly introduce the functional products and coproducts, since, as we will see, relational products require a different definition.

The *functional product* of two objects A and B is an object $A \times B$ and two arrows: $\pi_1 : A \leftarrow A \times B$ and $\pi_2 : B \leftarrow A \times B$. The object and the arrows must obey the following property: for each arrow $f : A \leftarrow C$ and $g : B \leftarrow C$ there exists an arrow $\langle f, g\rangle : A \times B \leftarrow C$, called 'split of f and g' such that

$$h = \langle f, g\rangle \equiv \pi_1 \cdot h = f \text{ and } \pi_2 \cdot h = g \tag{3.22}$$

for all $h : A \times B \leftarrow C$. This defines the universal property of products: $\langle f, g\rangle$ is the unique arrow that satisfies the property on the right-hand side of equation (3.22). This property can be represented by the following diagram:



where a dashed arrow represents its uniqueness.

Considering concrete categories, the functional product in **Fun** is given by the cartesian product and π_1 and π_2 are the projection functions. The arrow $\langle f, g\rangle$ would then be defined by

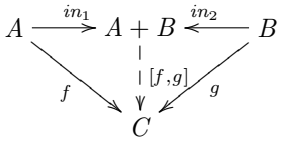
$$\langle f, g\rangle\ a = (f\ a, g\ a)$$

This definition however does not hold for **Rel**, since the pairing would have undefined values when the functions are partial. However, since $\mathbf{Rel} = \mathbf{Rel}^{op}$, functional products coincide with its dual operation, functional coproducts.

The *functional coproduct* is the dual operation of the functional product. So, products also consist of an object with two arrows, but with the source and target interchanged. The object is denoted by $A + B$ and the arrows $in_1 : A + B \leftarrow A$ and $in_2 : A + B \leftarrow B$. Given $f : C \leftarrow A$ and $g : C \leftarrow B$, the unique arrow $[f, g]$, called 'either f or g' is defined by the universal property:

$$h = [f, g] \equiv h \cdot in_1 = f \text{ and } h \cdot in_2 = g \tag{3.23}$$

for all $h : C \leftarrow A + B$. Once again, this information can be depicted in a diagram:



In the category **Fun**, functional coproducts are disjoint unions. In functional programing this would be represented by:

$$A + B ::= in_1\ A \mid in_2\ B$$

and the case operator defined by:

$$[f, g]\,(in_1\ a) = f\ a \text{ and } [f, g]\,(in_2\ a) = g\ a$$

Functional coproducts are represented in **Rel** the same way they are in **Fun**. Since **Rel** = $\mathbf{Rel}^{op}$, and functional coproducts are dual to functional products, this means functional products may also be represented by a disjoint union, i.e., $A \times B = A + B$. This poses a problem when representing types in a relational context, and thus, a new definition for relational products and coproducts is neded, which will surge in an allegorical setting.

3.3.2 Allegories and Relations

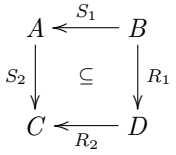
We will now generalize from functions to relations, whose advantages have already been stated. *Allegories* abstract relation algebras in the same way categories abstract function algebras. An allegory **A** is a category extended with three operators inspired by CR: inclusion, meet and converse.

Inclusion Two arrows of the same type can be compared by the partial order $\subseteq$, with respect to which composition is monotonic. In **Rel**, inclusion represents the set-theoretical inclusion, i.e.:

$$R \subseteq S \equiv \langle \forall\ a, b : a\ R\ b \Rightarrow a\ S\ b \rangle \tag{3.24}$$

Expressions on CCR are usually of the type $R \subseteq S$, called *inequations*, rather than equations. An expression of the type $R = S$ is usually decomposed into $R \subseteq S$ and $S \subseteq R$.

As in equations, diagrams are also used to represent inequations. In that case diagrams are said to *semi-commute*, which is denoted by a $\subseteq$ symbol. For instance, the expression $S_2 \cdot S_1 \subseteq R_2 \cdot R_1$ is depicted by:



Meet and converse In an allegory, for all arrows $R, S : A \leftarrow B$ there exists an arrow $R \cap S : A \leftarrow B$, called *meet* of R and S, whose universal property is, for all $X : A \leftarrow B$:

$$X \subseteq (R \cap S) \equiv (X \subseteq R) \text{ and } (X \subseteq S) \tag{3.25}$$

From this property we can derive that meet is commutative, associative and idempotent:

$$R \cap S = S \cap R \tag{3.26}$$

$$R \cap (S \cap T) = (R \cap S) \cap T \tag{3.27}$$

$$R \cap R = R \tag{3.28}$$

	Reflexive	Correflexive
$\ker R$	entire	injective
$\text{img}\, R$	surjective	simple

Table 3.1: Relation properties.

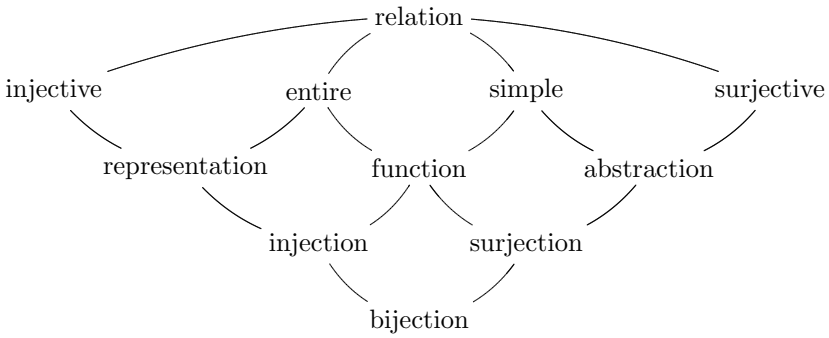


Figure 3.1: Relation taxonomy.

Finally, for every arrow $R : A \leftarrow B$ there exists an arrow $R^\circ : B \leftarrow A$ called the *converse* of R, which is an involution, order-preserving and contravariant:

$$(R^\circ)^\circ = R \tag{3.29}$$

$$R \subseteq S \equiv R^\circ \subseteq S^\circ \tag{3.30}$$

$$(R \cdot S)^\circ = S^\circ \cdot R^\circ \tag{3.31}$$

Reflexive and Correflexive Arrows Since we are using arrows to represent relations, they inherit some properties from relation algebras.

An arrow $R : A \leftarrow A$ is said to be reflexive if $id_A \subseteq R$. On the other hand, an arrow $R : A \leftarrow A$ is said to be correflexive if $R \subseteq id_A$, i.e., a fragment of the identity arrow.

Consider the definition of *kernel*, $\ker R \triangleq R^\circ \cdot R$, and *image*, $\text{img}\, R \triangleq R \cdot R^\circ$, of a relation. If the kernel of an arrow is reflexive ($id_A \subseteq \ker R$), then the arrow is *entire*. Also, if the image of an arrow is correflexive ($\ker R \subseteq id_A$), the arrow is *simple*. Correflexive kernels and reflexive images also induce relation properties, injectivity and surjectivity respectively. These properties are resumed in Table 3.1. Combinations of these properties define different classes of relations, as presented in Figure 3.1. For instance, arrows that are both entire and simple are functions. Lets quickly convert the definition os an injective relation back to PW to check its meaning:

$$\begin{array}{ll} & \ker R \subseteq id \\ \Leftrightarrow & \{\text{Def-ker}\} \\ & R \cdot R^\circ \subseteq id \\ \Leftrightarrow & \{\text{Def-Inclusion}\} \\ & \langle \forall\, x, y : x\, R \cdot R^\circ\, y \Rightarrow x\, id\, y \rangle \\ \Leftrightarrow & \{\text{Def-Composition, Def-Converse, Def-Id}\} \\ & \langle \forall\, x, y : \langle \exists\, k : x\, R\, k \wedge y\, R\, k \rangle \Rightarrow x = y \rangle \end{array}$$

Since correflexives are contained in the identity relation, they act as a filter, since whatever value is passed as input comes out as the output, if it belongs to the correflexive.

Range and Domain Every arrow $R : A \leftarrow B$ has associated two correflexives, the *domain* $\delta(R) : B \leftarrow B$ and the *range* $\rho(R) : A \leftarrow A$. Note that since $\delta(R) = \rho(R^\circ)$, the properties are easily interchangeable. The range may be defined by the universal law:

$$\delta(R) \subseteq X \equiv R \subseteq X \cdot R, \text{for all } X \subseteq id_A \tag{3.32}$$

However, it is usually directly defined by:

$$\delta(R) = (R \cdot R^\circ) \cap id \tag{3.33}$$

The domain definition and rules can be obtain by duality:

$$\rho(R) = (R^\circ \cdot R) \cap id \tag{3.34}$$

Tabular allegories Allegories do not fully define the characteristics of relations. However, one needs only to assume the existence of *tabulations* and a *unit* in the allegory to achieve set-theoretic relations. In particular, **Rel** is tabular and unitary.

An allegory is said to be *tabular* if for every arrow $R : A \leftarrow B$, there exist to functions f and g such that:

$$R = f \cdot g^\circ \tag{3.35}$$

$$(f^\circ \cdot f) \cap (g^\circ \cdot g) = id \tag{3.36}$$

Since relations can be seen as a subset C of a cartesian product $A \times B$, we have that projections $\pi_1 : A \leftarrow C$ and $\pi_2 : B \leftarrow C$ are tabulations of R.

A *unit* U is an object for which every arrow $U \leftarrow U$ is correflexive. Moreover, for every object A of the allegory, there exists an arrow $p_A : U \leftarrow A$. An allegory with a unit is said to be unitary. As a consequence, $p_A^\circ \cdot p_B$ is the largest arrow of type $A \leftarrow B$, which will be denoted by $\top_{A \leftarrow B} : A \leftarrow B$ and call *top*. Whenever the type can be deduced, we omit it and simply denote the relation as $\top$. In **Rel** the unit is the singleton set, and $\top : A \leftarrow B$ is the cartesian product $A \times B$.

Join We can extend the allegories with the *join* operator, denoted by $R \cup S$ for all $R, S : A \leftarrow B$, which represents the reunion of two relations. It is defined by the following universal law:

$$R \cup S \subseteq C \equiv (R \subseteq C) \text{ and } (S \subseteq C) \tag{3.37}$$

for all $X : A \leftarrow B$.

Like meet, join is associative, commutative and idempotency. However, unlike with meet, composition distributes over join:

$$R \cdot (S \cup T) = (R \cdot S) \cup (R \cdot T) \tag{3.38}$$

$$(S \cup T) \cdot R = (S \cdot R) \cup (T \cdot R) \tag{3.39}$$

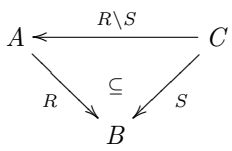
Allegories with the join operator, like **Rel**, are called *locally complete*, and give rise to other operators, which will be presented next.

Division The *left-division* and *right-division* operators are defined respectively by the following universal properties:

$$X \subseteq R \backslash S \equiv R \cdot X \subseteq S \tag{3.40}$$

$$X \subseteq R / S \equiv X \cdot R \subseteq S \tag{3.41}$$

Drawing the diagram of this definition, which semi-commutes, we get:



Since these are not common operators, we will resort to the point-wise definitions, which are more intuitive:

$$a\ R\backslash S\ b \equiv \langle \forall\ c : c\ R\ a \Rightarrow c\ S\ b \rangle \tag{3.42}$$

$$a\ R/S\ b \equiv \langle \forall\ c : a\ R\ c \Rightarrow b\ S\ c \rangle \tag{3.43}$$

Division allows us to represent universal quantifiers in point-free notation, much as the composition defines existential quantifiers.

Tabular allegories also allows us to define the implication and difference operators:

$$X \subseteq R \Rightarrow S \equiv R \cup X \subseteq S \tag{3.44}$$

$$X \subseteq S \cup R \equiv X - S \subseteq R \tag{3.45}$$

These kind of rules are called *Galois connections* (GC) [Ore44]. Although it is not the focus of our project, there's a different trend on program construction which is highly based on GC [Bac04]. In the theory of categories they are called *adjoints*. Note that the division operators were also be defined using GCs.

Relational product and coproduct As we've seen when presenting the categorical datatypes, the product and coproduct constructs are identical in **Rel**. So, in order to be useful constructs in an allegory, they must be defined in a different way. Without going into much detail, in a unitary tabular allegory, the *relational product* may be defined as:

$$\langle R, S \rangle \equiv (\pi_1^\circ \cdot R) \cap (\pi_2^\circ \cdot S) \tag{3.46}$$

The $\times$ operator is defined the usual way by:

$$R \times S \equiv \langle R \cdot \pi_1, S \cdot \pi_2 \rangle \tag{3.47}$$

The laws of categorical products do not hold for the allegorical product (unless we are dealing with function instead of relations). Instead, the have the following properties:

$$\pi_1 \cdot \langle R, S \rangle \equiv R \cdot \delta(S) \text{ and } \pi_2 \cdot \langle R, S \rangle \equiv S \cdot \delta(R) \tag{3.48}$$

$$\langle R, S \rangle^\circ \cdot \langle X, Y \rangle \equiv (R^\circ \cdot X) \cap (S^\circ \cdot Y) \tag{3.49}$$

The interpretation of the relational product is that $(a, b)\langle R, S\rangle c$ is defined when $a\ R\ c$ and $b\ S\ c$. If c is not in the domain of R or S, their product is not defined.

For the relational coproduct, we will need a *power allegory*, which is, very briefly, a allegory with power-sets. In that case, relational coproduct may be defined as:

$$[R, S] \equiv (R \cdot i_1^\circ) \cup (S \cdot i_2^\circ) \tag{3.50}$$

The $+$ operator is defined the same way as in the categorical setting:

$$R + S \equiv [i_1 \cdot R, i_2 \cdot S] \tag{3.51}$$

Chapter 4

RL to CCR translation

An automatic translation from RL to CCR will now be presented. Since CCR is characterization of RL without variables, this translation is actually a PW to PF transformation. The process is inspired both by the automatic translation proposed in [FPA04] and the heuristic method used in [Oli09], both of each we will present in these chapter. We will then present a new method which merges the two approaches, obtaining a still automatic translation, but with simpler results.

4.1 An automatic Alloy to FA translation

The first translation from RL to PF we will present is the one defined by Frias et al in [FPA04], with some type errors fixed, and missing cases of parametrized definitions inserted. This translation is defined for Alloy formulas, but for the moment only binary and unary relations will be considered. As such, variable application $a\,R\,b$ is not considered, but since this may be defined by `b in R.a` in Alloy, every RL formula may eventually be defined in Alloy notation, making this a possible RL to PF translation. Note also that although it was defined for FA, we present the translation with a categorical notation, for simplicity purposes.

The purpose of [FPA04] is to create a fully equational calculus, so they translate every Alloy formula to the form $\psi = \top$, with ψ a FA formula. This is supported by a result by Tarski [TG87], which states that every boolean combination of CR terms can be translated into a single equation of the form $\psi = \top$. So, starting with Alloy formulas without quantifiers, i.e., boolean combinations of Alloy terms, we apply the following rules to obtain an expression of the shape $R = \top$, which is still an Alloy expression.

$$\begin{aligned} R = S &\equiv R \cap S \cup (\top - R) \cap (\top - S) = \top \\ R \subseteq S &\equiv \top - (R - S) = \top \\ \neg(R = \top) &\equiv \top \cdot (\top - R) \cdot \top = \top \\ R = \top \wedge S = \top &\equiv R \cap S = \top \\ R = \top \vee S = \top &\equiv \top - ((\top - R) \cdot \top \cdot (\top - S)) = \top \\ R = \top \Rightarrow S = \top &\equiv \top \cdot (\top - R) \cdot \top \cup R = \top \end{aligned} \tag{4.1}$$

The expression R may contain m free variables $x_1, \ldots, x_n$. When these variables are instantiated with $b_1, \ldots, b_n$, the Alloy term is denoted by $R(b_1, \ldots, b_n)$. This term will be translated to a FA term $T_n(R)$ such that:

$$y\,R(b_1, \ldots, b_n)\,x \Leftrightarrow ((b_1, \ldots, b_n), y)\;T_n(R)\;((b_1, \ldots, b_n), x)$$

This means that in $T_n(R)$ all quantified variables are passed through the expression as arguments. When one of them is to be applied to a relation, a relation selects the appropriate variable

from the input. The relation X_i^n, with $1 \leq i \leq m$, relates an input $(b_1, \ldots, b_n)$, with the pairing associated to the left, with the output b_i:

$$X_i^n = \begin{cases} id & \text{if } n = 1 \\ \pi_2 & \text{if } m > 1 \wedge n = i \\ X_i^{n-1} \cdot \pi_1 & \text{if } n > 1 \wedge n > i \end{cases} \tag{4.2}$$

With that in mind, the translation of an Alloy term R is then performed by $T_n(R)$, where C is any relation, R and S Alloy terms and x_i a variable:

$$\begin{aligned} &T_n(C) = id_1 \times \cdots \times id_n \times C \\ &T_n(x_i) = \delta((X_i^n \cdot \pi_1) \cap \pi_2) \\ &T_n(R \cdot S) = \begin{cases} T_n(R) \cdot T_n(S) & \text{if } S, R \text{ binary} \\ \delta(T_n(R) \cdot T_n(S)) & \text{if } R \text{ unary} \\ \rho(T_n(R) \cdot T_n(S)) & \text{if } S \text{ unary} \end{cases} \\ &T_n(R \cap S) = T_n(R) \cap T_n(S) \\ &T_n(R \cup S) = T_n(R) \cup T_n(S) \\ &T_n(R - S) = T_n(R) \cap \overline{T_n(S)} \cap T_n(\top) \\ &T_n(R^\circ) = T_n(R)^\circ \end{aligned} \tag{4.3}$$

Note that every unary relation in Alloy is translated to a correflexive: such is the case of variable occurrence and composition of unary relations with binary ones. Concerning the variables, the term $\delta((X_i^n \cdot \pi_1) \cap \pi_2)$ is a correflexive which filters out inputs $((b_1, \ldots, b_n), x)$ where $x \neq b_i$, i.e., the abstract input x is forced to take the value of b_i. Composition of an unary relation $S : B$, with a binary one, $R : A \leftarrow B$ is translated to $\rho(R \cdot S)$, a correflexive of type $A \leftarrow A$. Composition of two unary relations is not valid, since relations with arity 0 are not allowed in Alloy.

Using the rules from (4.1), and then applying T_n, we obtain an equation of the shape $\psi = \top$, where ψ is already a FA term. We will now consider the quantifiers, assuming they are applied to formulas of that shape.

Note that from (4.1), if there are m free variables, we get:

$$\begin{aligned} &(b_1, \ldots, b_n, y)\ T_n(R)\ (b_1, \ldots, b_n, x) \\ \Leftrightarrow &\{\text{Def-}\pi\} \\ &y\ \pi_2 \cdot T_n(R)\ (b_1, \ldots, b_n, x) \\ \Leftrightarrow &\{\text{Def-Split}\} \\ &(b_1, \ldots, b_n, x, y)\ \langle id, \pi_2 \cdot T_n(R) \rangle\ (b_1, \ldots, b_n, x) \\ \Leftrightarrow &\{\text{Def-Range,Def-}\top\} \\ &c\ \top \cdot \rho(\langle id, \pi_2 \cdot T_n(R) \rangle)\ (b_1, \ldots, b_n, x, y) \end{aligned}$$

The input and output of the term $T_n(R)$ are abstracted by x and y, which must be universally quantified. This translation pushes the output y to the right hand side, transferring all variable manipulation to that side. As such, the left-hand side is disregarded, which is represented by the composition of a $\top$ relation on that side. That transformation will be represented by:

$$\dagger_n (R) \triangleq \begin{cases} \top \cdot \rho(\langle id, T_n(R) \rangle) & \text{if } m = 0 \\ \top \cdot \rho(\langle id, \pi_2 \cdot T_n(R) \rangle) & \text{if } m > 0 \end{cases} \tag{4.4}$$

We now define a relational term $\exists_i^n$, which will cut the existential quantifiers from the formula:[1]

$$\exists_i^n \triangleq \langle X_1^n, \ldots, X_{i-1}^n, \top, X_i^n, \ldots, X_n^n \rangle \tag{4.5}$$

[1] $\langle R_1, \ldots, R_n \rangle$ is an abuse of notation, meaning $\langle \langle R_1, \ldots \rangle \ldots \rangle, R_n \rangle$.

When $m = 0$, $\exists_i^0 = \top$. Note that the composition of this operator will introduce an existential quantified variable at position i, and the use of $\top$ states that any value may be assigned to variable x_i. As an example:

$$c\,R \cdot \langle X_1^2, \top, X_2^2 \rangle\,(x_1, x_3) \Leftrightarrow \langle \exists\, x_2 :: c\,R(x_1, x_2, x_3)\,\rangle$$

Finally, since the existential and universal quantifiers are inter-definable, we can define the universal quantification as:

$$\forall_i^n\ R \triangleq \overline{\overline{R} \cdot \exists_i^n} \tag{4.6}$$

The double negation will transform the existential quantification into a universal quantification.

Now that the translation of quantifiers is defined, we can define the top-level rules for the translation of Alloy formulas into FA:

$$\begin{aligned} T'_n(R = \top) &= \forall_{n+2}^{n+1}\, \forall_{n+1}^n\, \dagger_n(R) \\ T'_n(\exists\, x_i : R) &= T'_{n+1}(R) \cdot \exists_i^n \\ T'_n(\forall\, x_i : R) &= \forall_i^n\ T'_{n+1}(R) \end{aligned} \tag{4.7}$$

The change from n to $n+1$ are due to the introduction of a new variable by the quantifiers.

Finally, for any Alloy formula F, we define its transformation into FA by $T'_0(F) = \top$.

The resulting expressions of this translation are usually very complex. Even though the goal was to verify the specifications by theorem provers (in this case, PVS), the results were not very positive, and the authors eventually abandoned this approach [FPM07]. In our case, it goes against one of the benefits of using PF calculus: obtaining simple and easily manipulable expressions.

Let's see the resulting translation of an example. A very simple expression, with only two quantifiers:

$$\langle \forall\, a :: \langle \exists\, c :: c \subseteq R \cdot a \wedge c \subseteq S \cdot a \rangle \rangle \tag{4.8}$$

The first step is transforming the boolean combination inside the quantifications into the form $R = \top$:

$$\begin{aligned} & c \subseteq R \cdot a \wedge c \subseteq S \cdot a \\ \Leftrightarrow & \{(4.3)\} \\ & \top - (c - R \cdot a) = \top \wedge \top - (c - S \cdot a) = \top \\ \Leftrightarrow & \{(4.3)\} \\ & (\top - (c - R \cdot a)) \cap (\top - (c - S \cdot a)) = \top \end{aligned}$$

Next, we apply the T_2 transformation to R:

$$\begin{aligned} & T_2((\top - (c - R \cdot a)) \cap (\top - (c - S \cdot a))) \\ \Leftrightarrow & \{\text{Def-}T_n\} \\ & T_2(\top - (c - R \cdot a)) \cap T_2(\top - (c - S \cdot a)) \\ \Leftrightarrow & \{\text{Def-}T_n\} \\ & T_2(\top) \cap \overline{T_2(c - R \cdot a)} \cap \overline{T_2(c - S \cdot a)} \\ \Leftrightarrow & \{\text{Def-}T_n\} \\ & T_2(\top) \cap \overline{T_2(c) \cap \overline{T_2(R \cdot a)} \cap T_2(\top)} \cap \overline{T_2(c) \cap \overline{T_2(S \cdot a)} \cap T_2(\top)} \\ \Leftrightarrow & \{\text{Def-}T_n\} \\ & T_2(\top) \cap \overline{T_2(c) \cap \overline{\rho(T_2(R) \cdot T_2(a))} \cap T_2(\top)} \cap \overline{T_2(c) \cap \overline{\rho(T_2(S) \cdot T_2(a))} \cap T_2(\top)} \\ \Leftrightarrow & \{\text{Def-}T_n, \text{Def-}X_i^n\} \\ & id \times id \times \top \cap \overline{\delta(\pi_2 \cdot \pi_1 \cap \pi_2) \cap \overline{\rho(id \times id \times R \cdot \delta(\pi_1 \cdot \pi_1 \cap \pi_2))} \cap id \times id \times \top} \cap \\ & \qquad \overline{\delta(\pi_2 \cdot \pi_1 \cap \pi_2) \cap \overline{\rho(id \times id \times S \cdot \delta(\pi_1 \cdot \pi_1 \cap \pi_2))} \cap id \times id \times \top} \end{aligned}$$

We will call this resulting FA expression ϕ. Now, applying T'_0 to the whole expression we get:

$$
\begin{aligned}
& T'_0(\forall\, a :: \exists\, c :: c\ R\ a \wedge c\ S\ a) = \top \\
\Leftrightarrow & \{\text{Def-}T'_n\} \\
& \forall_1^0(\forall_3^2 \forall_4^3 \dagger_2(c\ R\ a \wedge c\ S\ a)) \cdot \exists_2^1) = \top \\
\Leftrightarrow & \{\text{Def-}\exists_i^n, \text{Def-}\forall_i^n\} \\
& \overline{\overline{\overline{\dagger_2(c\ R\ a \wedge c\ S\ a)} \cdot \langle X_1^3, X_2^3, X_3^3, \top\rangle \cdot \langle X_1^2, X_2^2, \top\rangle} \cdot \langle X_1^1, \top\rangle} \cdot \top = \top \\
\Leftrightarrow & \{\text{Def-}\dagger_n\ (R), \text{Def-}X_i^n\} \\
& \overline{\overline{\overline{\top \cdot \rho(\langle id, \pi_2 \cdot \phi\rangle)} \cdot \langle \pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1, \pi_2, \top\rangle \cdot \langle \pi_1, \pi_2, \top\rangle} \cdot \langle id, \top\rangle} \cdot \top = \top
\end{aligned}
\tag{4.9}
$$

As can be seen, an expression as simple as this one results in a FA expression of such complexity that not only it is impossible to manipulate, but also its original meaning is completely lost, even for the trained eye.

4.2 Strategic Rewriting

In the following sections, our translations will be presented as a *strategic rewriting* [LV02] process. In strategic rewriting, a set of rewriting rules are defined, which are then combined using strategic combinators to produce more powerful rules

A term rewriting *rule* is defined with pattern matching, and thus may fail. We will use the notation $R \rightsquigarrow S$ to denote that a term of the shape R is transformed into S. Application of the rule to terms of different shape fail.

Rules may be combined with strategic combinators to produce complex transformations. The first one we will present is the *sequence* combinator, denoted by $\triangleright$, which given two rules, applies the second to the result of the first, if it was applied successful. The *choice* combinator $\oslash$ of two rules tries to perform the first, and if it fails, applies the second. The *many* combinator applies a given rule repetitively until it fails. Note that a rule of the form *many r* never fails.

These operators combine rules at term level, but do not descend into the expression. For that purpose, we define the *once* operator, which applies the rule once somewhere inside the expression.

Throughout the rest of the thesis we will assume that a rule defined as:

$$
\begin{aligned}
R_1 &\rightsquigarrow S_1 \\
R_2 &\rightsquigarrow S_2 \\
\ldots &\rightsquigarrow \ldots \\
R_n &\rightsquigarrow S_n
\end{aligned}
$$

represents the choice of each line, in the order presented, i.e., $R_1 \rightsquigarrow S_1 \oslash R_2 \rightsquigarrow S_2 \oslash \cdots \oslash R_n \rightsquigarrow S_n$.

4.3 Eindhoven quantifications

Throughout the rest of the thesis, we will present universal and existential quantifications using the Eindhoven notation [Bac04]:

$$
\begin{aligned}
&\langle \forall\, x : R : S \rangle \\
&\langle \exists\, x : R : S \rangle
\end{aligned}
$$

This notation represents the expression "for all x in range R, T holds", in the case of the universal quantification. If range is empty, the quantification is simply denoted by $\langle \forall\, x :: R \rangle$.

Although they provide only a different notation for the quantifications, they provide better readability and easily manipulation of quantifications, since to this notation is allied a set of rules about them, which we will call Eindhoven laws. We present in Table 4.1 some of the most used.

Name	Law
De Morgan	$\neg\langle \forall\, x : R : S\rangle \equiv \langle \exists\, x : R : \neg S\rangle$ $\neg\langle \exists\, x : R : S\rangle \equiv \langle \forall\, x : R : \neg S\rangle$
Trading	$\langle \forall\, x : R \wedge S : T\rangle \equiv \langle \forall\, x : R : S \Rightarrow T\rangle$ $\langle \exists\, x : R \wedge S : T\rangle \equiv \langle \exists\, x : R : S \wedge T\rangle$
One-point	$\langle \forall\, x : x = e : S\rangle \equiv S[x := e]$ $\langle \exists\, x : x = e : S\rangle \equiv S[x := e]$
Nesting	$\langle \forall\, x : R : \langle \forall\, y : S : T\rangle\rangle \equiv \langle \forall\, x, y : R \wedge S : T\rangle$ $\langle \exists\, x : R : \langle \forall\, y : S : T\rangle\rangle \equiv \langle \exists\, x, y : R \wedge S : T\rangle$
Splitting	$\langle \forall\, x : R : \langle \forall\, y : S : T\rangle\rangle \equiv \langle \forall\, y : \langle \exists\, x : R : S\rangle : T\rangle$ $\langle \exists\, x : R : \langle \exists\, y : S : T\rangle\rangle \equiv \langle \exists\, y : \langle \exists\, x : R : S\rangle : T\rangle$

Table 4.1: Eindhoven quantifier laws.

4.4 An improved automatic RL to CCR translation

In the previous section, we presented an automatic translation from Alloy to FA, which resulted in very complex expressions. With expressions that complex, the goal of using PF expressions is lost: easy understanding and manipulation. So, inspired in that translation, we define a new translation from RL to CCR which is still automatic, but with a simpler result. Note that although omitted for simplicity purposes, the transformations are strongly typed.

Like the previous translation, our translation reduces the formula to the shape $\top \subseteq R$, which is actually the same as $R = \top$, as for all R, $R \subseteq \top$. Also, we convert the input of the relations to a tuple containing all quantified variables, from which the desired variable is then selected. However, the way these two goals are achieved differ significantly from the original, which results in a simpler translation.

First, note that in (4.5) when inserting a new variable, it may be inserted in any position of the tuple containing the variables. However, in the presentation in [FPA04], the variable is always inserted last position. As such, $\exists_i^n$ may be always defined as $\langle id, \top\rangle$, except when $i = 1$, when it is simply $\top$, which is considerably simpler. Note that the parameters n and i, no longer need to be passed to this operator.

Second, in the presented translation, all variable occurrences were directly translated to a correflexive, $\delta((X_i \cdot \pi_1) \cap \pi_2)$, which forced the input to take the value of the desired variable. This way, Alloy navigational expressions could be directly translated, since Alloy variables may appear "composed" anywhere inside a term. This also meant that all variables needed to be passed through all relations, which resulted in the translation of every relation R into $id_1 \times \cdots \times id_n \times R$. In our case, our approach is completely different. Since we assume we are not dealing with Alloy expressions, but rather RL, those navigations were already previously expanded (see chapter 5), so we are dealing with boolean combinations of applications. As such, our process will consist on reducing the input and output of those applications to the same variables, using the selection operator X_i^n.

Third and last, the presented translation transformed every boolean combination into the shape $R = \top$, which implied using the rules from (4.1), highly increasing the complexity of the expression. In our case, although in the end we also achieve the form $R = \top$, we do not enforce that shape in the inner terms. Instead, we just reduce the output and input of every CCR term to the same variables, and then apply the definition of the intersection and join operators.

In the end, our default translation has little similarities with the original one. We assume that the initial formula is previously treated to remove some operators, i.e., variable equality is translated to identity and implication to its definition as a disjunction, and redundant *true* and *false* constants inside conjunctions and disjunctions. If the initial formula is a boolean combination, the following translation will be applied to each element. In the end, we obtain a boolean combination of inequations.

Insert external variables If the expressions is not already completely PF, insert two external universal quantifications in the expression, which we will call x and y.

$$\langle \forall\, x, y :: R \rangle \tag{4.10}$$

These two variables will abstract the input and output of the final expression, of the form $\top \subseteq R$ and as such will *not belong to the variable tuple*. Introducing quantifications of variables which do not occur in the expression does not change its meaning, as long as the type of the quantified variable is not empty. It seems unnecessary to introduce the external quantifications so earlier in the translation, but in the next section we will present some simplifications that must be applied right after the introduction of the external variables, and before the remaining steps.

This rule is only applied once and at the top level, so this step is defined as:

$$\mathit{insExtVars} \triangleq (4.10)$$

Remove universal quantifications Quantifiers will be removed by composing the expression with $\langle id, \top \rangle$, which works only for existential quantifiers. As such, we transform every universal quantifier, *except the two introduced in the previous step*, into an existential one, using the De Morgan's laws:

$$k \notin \{x, y\} \Rightarrow \langle \forall\, k : R : S \rangle \rightsquigarrow \neg \langle \exists\, k : R : \neg S \rangle \tag{4.11}$$

Also, in order to remove the range from the quantifications, we apply the existential trading rule on every existential quantification:

$$\langle \exists\, k : R : S \rangle \rightsquigarrow \langle \exists\, k :: R \wedge S \rangle \tag{4.12}$$

This way there is no longer need to be concerned with the range of the quantifier, and since the existential trading only adds a conjunction, it does not increase the complexity of the expression.

This step is thus defined as:

$$\mathit{remUniversal} \triangleq \mathit{many}\,(\mathit{once}\,(4.11) \oslash \mathit{once}\,(4.12))$$

Uniform variable applications Our goal is to convert the input of all applications to the variable tuple, shifting all variable manipulation to the input, thus uniforming it and the output. In order to do so, at every variable application, we generalize the input into the respective variable tuple, using the selection operator X_i^n:

$$b_j\ R\ b_i \rightsquigarrow b_j\ R \cdot X_i^n\,(b_1, \ldots, b_n)$$

The variable tuple contains the n quantified variables *at that level of the expression* except the two most external universal quantifications (x and y), i.e., all existential quantified variables, by the order they occur, associated to the left. However, if the input is a tuple, splits of selection operators are used to generalize it to the variable tuple:

$$b_j\ R\,(b_i, \ldots, b_{i'}) \rightsquigarrow b_j\ R \cdot \langle X_i^n, \ldots, X_{i'}^n \rangle\,(b_1, \ldots, b_n)$$

The output is converted to a new existential quantified variable, related to the variable tuple by the selection operator X_i^n. This way, all variable treatment is shifted to the input of the terms, and the output may be disregarded:

$$b_j\ R \cdot \langle X_i^n, \ldots, X_{i'}^n \rangle\,(b_1, \ldots, b_n) \rightsquigarrow \langle \exists\, k :: k\ X_j^n\,(b_1, \ldots, b_n) \wedge k\ R \cdot \langle X_i^n, \ldots, X_{i'}^n \rangle\,(b_1, \ldots, b_n) \rangle$$

Again, if the output is a tuple, splits of selection operators are used to generalize it to the variable tuple:

$$\begin{aligned}(b_j, \ldots, b_{j'})\ R \cdot \langle X_i^n, \ldots, X_{i'}^n \rangle\,(b_1, \ldots, b_n) \rightsquigarrow \\ \langle \exists\, k :: k\ \langle X_j^n, \ldots, X_{j'}^n \rangle\,(b_1, \ldots, b_n) \wedge k\ R \cdot \langle X_i^n, \ldots, X_{i'}^n \rangle\,(b_1, \ldots, b_n) \rangle\end{aligned}$$

Once on that shape, apply the definition of intersection inside the existential quantifiers created in the previous step:

$$\langle \exists\, k :: k\, \langle X_j^n, \ldots, X_{j'}^n \rangle\, (b_1, \ldots, b_n) \wedge k\, R \cdot \langle X_i^n, ..., X_{i'}^n \rangle\, (b_1, \ldots, b_n) \rangle \rightsquigarrow \\ \langle \exists\, k :: k\, (\langle X_j^n, \ldots, X_{j'}^n \rangle \cap R \cdot \langle X_i^n, ..., X_{i'}^n \rangle)\, (b_1, \ldots, b_n) \rangle$$

Finally, the existential quantifier introduced is cut by composing the term with a $\top$ relation. The output becomes one of the *external universally quantified variables*, concretely x:

$$\langle \exists\, k :: k\, (\langle X_j^n, \ldots, X_{j'}^n \rangle \cap R \cdot \langle X_i^n, ..., X_{i'}^n \rangle)\, (b_1, \ldots, b_n) \rangle \rightsquigarrow \\ x\, \top \cdot (\langle X_j^n, \ldots, X_{j'}^n \rangle \cap R \cdot \langle X_i^n, ..., X_{i'}^n \rangle)\, (b_1, \ldots, b_n)$$

Summarizing all those transformations in a single rule, every variable applications is uniformed by:

$$(b_j, \ldots, b_{j'})\, R\, (b_i, \ldots, b_i') \rightsquigarrow x\, \top \cdot (\langle X_j^n, ..., X_{j'}^n \rangle \cap R \cdot \langle X_i^n, ..., X_{i'}^n \rangle)\, (b_1, \ldots, b_n) \tag{4.13}$$

$$\mathit{uniformIO} \triangleq \mathit{many}\, (\mathit{once}\, (4.13))$$

Reduce boolean combinations By now, every formula without quantifiers is a boolean combination of applications $x\, Z\, v$ with x one of the externally quantified variables and v the variable tuple. As such, the definitions of RL operators join, intersection and complement is applied to transform the boolean combinations into a single CCR term:

$$\begin{aligned} x\, R\, v \wedge x\, S\, v &\rightsquigarrow x\, R \cap S\, v \\ x\, R\, v \vee x\, S\, v &\rightsquigarrow x\, R \cup S\, v \\ \neg (x\, R\, v) &\rightsquigarrow x\, \overline{R}\, v \end{aligned} \tag{4.14}$$

$$\mathit{reduceBC} \triangleq \mathit{once}\, (4.14)$$

Remove quantifications While $n > 1$, we compose the terms with the relation $\langle id, \top \rangle$ on the right side to cut the right-most existential quantified variable from the variable tuple:

$$\langle \exists\, b_n :: x\, S\, (b_1, \ldots, b_n) \rangle \rightsquigarrow x\, S \cdot \langle id, \top \rangle\, (b_1, \ldots, b_{n-1}) \tag{4.15}$$

If $n = 1$, the term is composed with $\top$ to remove the last existential quantification, using the second external universal variable, y, as input:

$$\langle \exists\, b_1 :: x\, S\, b_1 \rangle \rightsquigarrow x\, S \cdot \top\, y \tag{4.16}$$

$$\mathit{remQuant} \triangleq \mathit{many}(\mathit{once}\, (4.15) \oslash \mathit{once}\, (4.16))$$

If the expression is still a boolean expression, we return to the reduction of the boolean quantifications. We cycle between these two steps until the expression has only the external quantifiers left. Since we previously converted the input and output to the same variable, we know that point will eventually be achieved.

Drop external variables If the expression is of the shape $\langle \forall\, x, y :: x\, S\, y \rangle$, we apply the definition of inclusion and drop the two external variables:

$$\begin{aligned} \langle \forall\, x, y :: x\, S\, y \rangle &\rightsquigarrow \top \subseteq S \\ \langle \forall\, x, y : x\, R\, y : x\, S\, y \rangle &\rightsquigarrow R \subseteq S \end{aligned} \tag{4.17}$$

$$\mathit{dropExtVars} \triangleq (4.17)$$

This rule completely drops the variables, achieving a fully PF inequation. Note that since the default translation removes all ranges, the second case will never be applied. However, once we start applying the heuristics defined in the following sections, that situation might occur.

The full translation is then defined as:

$$insExtVars \triangleright remUniversal \triangleright uniformIO \triangleright many\,(dropExtVars \oslash (reduceBC \oslash remQuant))$$

Let's see how it works with the example expression (4.8):

$$\begin{aligned}
&\langle \forall\, a :: \langle \exists\, b : a\ R\ b : a\ S\ b \rangle \rangle \\
\Leftrightarrow&\{insExtVars\} \\
&\langle \forall\, x, y :: \langle \forall\, a :: \langle \exists\, b : a\ R\ b : a\ S\ b \rangle \rangle \rangle \\
\Leftrightarrow&\{remUniversal\} \\
&\langle \forall\, x, y :: \neg\langle \exists\, a :: \neg\langle \exists\, b :: a\ R\ b \wedge a\ S\ b \rangle \rangle \rangle \\
\Leftrightarrow&\{uniformIO\} \\
&\langle \forall\, x, y :: \neg\langle \exists\, a :: \neg\langle \exists\, b :: x\ \top \cdot (\pi_1 \cap R \cdot \pi_2)\,(a,b) \wedge x\ \top \cdot (\pi_1 \cap S \cdot \pi_2)\,(a,b) \rangle \rangle \rangle \\
\Leftrightarrow&\{reduceBC\} \\
&\langle \forall\, x, y :: \neg\langle \exists\, a :: \neg\langle \exists\, b :: x\ \top \cdot (\pi_1 \cap R \cdot \pi_2) \cap \top \cdot (\pi_1 \cap S \cdot \pi_2)\,(a,b) \rangle \rangle \rangle \\
\Leftrightarrow&\{remQuant\} \\
&\langle \forall\, x, y :: \neg\langle \exists\, a :: \neg x\ (\top \cdot (\pi_1 \cap R \cdot \pi_2) \cap \top \cdot (\pi_1 \cap S \cdot \pi_2)) \cdot \langle id, \top \rangle\ a \rangle \rangle \\
\Leftrightarrow&\{reduceBC\} \\
&\langle \forall\, x, y :: \neg\langle \exists\, a :: x\ \overline{(\top \cdot (\pi_1 \cap R \cdot \pi_2) \cap \top \cdot (\pi_1 \cap S \cdot \pi_2)) \cdot \langle id, \top \rangle}\ a \rangle \rangle \\
\Leftrightarrow&\{remQuant\} \\
&\langle \forall\, x, y :: \neg x\ \overline{(\top \cdot (\pi_1 \cap R \cdot \pi_2) \cap \top \cdot (\pi_1 \cap S \cdot \pi_2)) \cdot \langle id, \top \rangle} \cdot \top\ y \rangle \\
\Leftrightarrow&\{reduceBC\} \\
&\langle \forall\, x, y :: x\ \overline{\overline{(\top \cdot (\pi_1 \cap R \cdot \pi_2) \cap \top \cdot (\pi_1 \cap S \cdot \pi_2)) \cdot \langle id, \top \rangle} \cdot \top}\ y \rangle \\
\Leftrightarrow&\{dropExtVars\} \\
&\top \subseteq \overline{\overline{(\top \cdot (\pi_1 \cap R \cdot \pi_2) \cap \top \cdot (\pi_1 \cap S \cdot \pi_2)) \cdot \langle id, \top \rangle} \cdot \top}
\end{aligned}$$

As can be seen, this expression is much simpler than the one resulting from the original automatic translation (4.9). However, it is still far from the simplicity that can be achieved in PF expressions.

4.5 An heuristic translation from RL to CCR

The translation we presented automatically transforms RL expressions to CCR. However, the result is still not very simple. Our reason to use CCR expressions, a PF framework, is to be able to easily understand and manipulate expression, since PF expressions can be very simple. The method we will now present is based on the calculus style of José Nuno Oliveira [Oli09], where the PF transform is seen as the Laplace transform: by transforming PW expressions to simple PF expressions, formula manipulation and solving complex problems may become easier, and at the end, the result can be transformed back to PW. This method is inspired by the PF RL defined by Bird and Moor [BdM96], and the judicious use of Galois connections from the school of Eindhoven [Bac04].

Unlike the previous presented in the previous section, it is not a mechanic translation, but rather an heuristic process. Due to this "human" factor, the resulting expressions are usually very simple and suitable for manual proofs The drawback is that, that same human factor is an obstacle for the creation of a fully automatic translation method. As such, our goal will be to define a set

PW	PF
$\langle \forall\, a, b : a\, R\, b : a\, S\, b \rangle$	$R \subseteq S$
$\langle \exists\, c : a\, R\, c : c\, S\, b \rangle$	$a\, R \cdot S\, b$
$\langle \forall\, c : c\, R\, a : c\, S\, b \rangle$	$a\, R \backslash S\, b$
$\langle \forall\, c : a\, R\, c : b\, S\, c \rangle$	$a\, R / S\, b$
$a\, R\, c \wedge b\, S\, c$	$(a, b)\, \langle R, S \rangle\, c$
$a\, R\, c \wedge b\, S\, d$	$(a, b)\, R \times S\, (c, d)$
$a\, R\, b \wedge a\, S\, b$	$a\, R \cap S\, b$
$a\, R\, b \vee a\, S\, b$	$a\, R \cup S\, b$
True	$b \top a$
False	$b \bot a$

Table 4.2: Some of the rules used for the PF-transform.

of rules which can be used to simplify the translation presented in the previous chapter, which we will now call the *default translation*.

Another essential difference between this translation and the previous, is that this one was developed in a categorical setting taking full advantage of the existing operators.

The restriction of the first translation that every formula had to be of the form $R = \top$ does not exist in this case, since the goal is not to achieve an equational calculus. Basic expressions are of the shape $R \subseteq S$, inequations, and this fact alone causes the resulting PF expressions to be much simpler than the ones using the first method, as the rules (4.1) do not need to be applied. Also, a single PW expression may result in multiple inequations. In the end, the model consists of a boolean combination of inequations.

Since the goal of the translation is to achieve expressions of the form $R \subseteq S$, ultimately, the definition of the inclusion operator 3.24 is the rule that completely drops the variables. So, the goal of the method is actually achieving an expression of the form $\langle \forall\, a, b : a\, R\, b : a\, S\, b \rangle$, or a boolean combination of those. The use of Eindhoven quantifiers is essential to that process, since not only they allow us to remove many quantifiers, but they also allow us the control the range of the quantifiers.

Internal quantifiers are cut either by the composition operator (3.20), if existential, or the division operators, (3.43) and (3.42), if universal. Other than this, the method consists of the application of the basic rules of Table 4.2, or any RL rule for boolean combinations which might help achieve that goal.

Another characteristic of this method is the use of the relation taxonomy defined in 3.3.2 in order to apply more particular rules. For instance, functions benefit from many properties that relations do not. In particular, the use of Galois connections highly depends on the classification of the relations. More than simplifying, GCs provide a way to easily prove properties. Since at the moment that is not our main concern, we will not deepen our presentation on these laws.

A consequence of this approach, completely driven towards simplicity, is that it highly depends on the richness of properties about the operators. As such, operators with few properties are to be avoided. Particularly, we refer the complement operation which is to be avoided in this process, due to its lack of useful properties.

Although at first sight the absence of a mechanical transformation may create the impression that this method is not feasible, it can be applied to most expressions with very good results. Let's take the very basic example (4.8) provided in the previous section:

$$
\begin{aligned}
& \langle \forall\, a :: \langle \exists\, c :: c\, R\, a \wedge c\, S\, a \rangle \rangle \\
\Leftrightarrow & \{\text{Def-Composition, Def-Converse}\} \\
& \langle \forall\, a :: a\, R^{\circ} \cdot S\, a \rangle \\
\Leftrightarrow & \{\text{Eindhoven: One-Point}\} \\
& \langle \forall\, a, b : a = b : a\, R^{\circ} \cdot S\, b \rangle \\
\Leftrightarrow & \{\text{Def-Inclusion}\}
\end{aligned}
$$

$\wedge$ rules	$\vee$ rules
$A \wedge true \rightsquigarrow A$	$A \vee false \rightsquigarrow A$
$A \wedge false \rightsquigarrow false$	$A \vee true \rightsquigarrow true$
$A \wedge A \rightsquigarrow A$	$A \vee A \rightsquigarrow A$
$\neg(A \wedge B) \rightsquigarrow \neg A \vee \neg B$	$\neg(A \vee B) \rightsquigarrow \neg A \wedge \neg B$
$\neg$ rules	$\Rightarrow$ rules
$\neg\neg A \rightsquigarrow A$	$\neg A \vee B \rightsquigarrow A \Rightarrow B$
$\neg true \rightsquigarrow false$	$false \Rightarrow A \rightsquigarrow true$
$\neg false \rightsquigarrow true$	$true \Rightarrow A \rightsquigarrow A$
	$A \Rightarrow true \rightsquigarrow true$
= rules	$A \Rightarrow false \rightsquigarrow \neg A$
$a = b \rightsquigarrow a\ id\ b$	$A \Rightarrow (B \Rightarrow C) \rightsquigarrow (A \wedge B) \Rightarrow C$
Eindhoven	
$\langle \forall\, a : R : S \Rightarrow T \rangle \rightsquigarrow \langle \forall\, a : R \wedge S : T \rangle$	
$\langle \exists\, a : R : S \rangle \rightsquigarrow \langle \forall\, a :: R \wedge S \rangle$	
$\langle \forall\, a : R : \langle \forall\, b : S : T \rangle \rangle \rightsquigarrow \langle \forall\, a, b : R \wedge S : T \rangle$	
$\langle \exists\, a : R : \langle \exists\, b : S : T \rangle \rangle \rightsquigarrow \langle \exists\, a, b : R \wedge S : T \rangle$	
$\langle \exists\, x_1, \ldots, c, \ldots, x_k :: c\ id\ x_i \wedge Z \rangle \rightsquigarrow \langle \exists\, x_1, \ldots, x_k :: Z[c := x_i] \rangle$	
$\langle \exists\, x_1, \ldots, c, \ldots, x_k :: c\ id\ x_i \rangle \rightsquigarrow \langle \exists\, x_1, \ldots, x_k :: true \rangle$	
$\langle \forall\, x_1, \ldots, c, \ldots, x_k : c\ id\ x_i \wedge W : Z \rangle \rightsquigarrow \langle \forall\, x_1, \ldots, x_k :: W[c := x_i] \Rightarrow Z[c := x_i] \rangle$	
$\langle \forall\, x_1, \ldots, c, \ldots, x_k : c\ id\ x_i : Z \rangle \rightsquigarrow \langle \forall\, x_1, \ldots, x_k :: Z[c := x_i] \rangle$	

Table 4.3: FOL transformations (*FOLrules*).

$$id \subseteq R^\circ \cdot S \tag{4.18}$$

Comparing the result with (4.9), the resulting expression using the translation from [FPA04], the difference in complexity is enormous.

4.6 Mechanizing heuristic simplifications

Moving back to our default translation, we will define a set of rules which are applied *before* the translation kicks in, and a set of rules to simplify the result, applied *after* the translation.

The first set of rules we define for the simplification of the formula are the FOL rules, which are represented in Table 4.3 and denoted by *FOLrules*, and include the properties of boolean operators, and some of the Eindhoven laws. The boolean properties dispense presentation, and perform basic simplifications. Eindhoven laws will remove redundant quantifications and reduce the complexity of the expressions. We apply the existential and universal trading rules, albeit in different directions, and the nesting laws. Applying the universal trading from the left to the right removes a implication and allows us to benefit from rules that manipulate the range of universal quantifications. Also, since final expressions are of the form $R \subseteq S$, it allows us to spread the complexity through both sides of the inequation. Since applying the existential trading does not increase nor decrease the complexity of the expression and there are no properties that benefit from the existence of the existential range, we apply it from the right to the left side, disregarding the range of existential quantifiers. As for the nesting laws, they are always applied from the right to the left side, as that transformation pulls the quantifications outwards. The one-point laws are also applied remove existential quantifications.

Note that although omitted for simplicity purposes, the rules consider the commutativity and associativity of the operators. For instance, the idempotency rule for conjunctions, $R \wedge R \rightsquigarrow R$, searches through the conjunction of any elements, since the conjunction is associative and commutative. Also, when a table as a rules, for instance *FOLrules*, we assume the *choice* of all the rules there presented is applied *once*.

$R = S \rightsquigarrow R \subseteq S \wedge S \subseteq R$	$a\ R\ b \wedge a\ S\ b \rightsquigarrow a\ R \cap S\ b$
$x = y \rightsquigarrow x\ id\ y$	$a\ R\ b \wedge b\ S\ a \rightsquigarrow a\ R \cap S^\circ\ b$
$R \Rightarrow S \rightsquigarrow \neg R \vee S$	$a\ R\ b \vee a\ S\ b \rightsquigarrow a\ R \cup S\ b$
$x \perp y \rightsquigarrow false$	$a\ R\ b \vee b\ S\ a \rightsquigarrow a\ R \cup S^\circ\ b$
$x \top y \rightsquigarrow true$	$a\ R\ c \wedge b\ S\ c \rightsquigarrow (a, b)\ \langle R, S\rangle\ c$
$\neg a\ R\ c \rightsquigarrow a\ \overline{R}\ c$	$a\ R\ c \wedge c\ S\ b \rightsquigarrow (a, b)\ \langle R, S^\circ\rangle\ c$
	$c\ R\ a \wedge c\ S\ b \rightsquigarrow (a, b)\ \langle R^\circ, S^\circ\rangle\ c$
$c \notin vars(Z) \Rightarrow \langle \exists\, x_1, \ldots, c, \ldots, x_k :: x_i\ R\ c \wedge c\ S\ x_j \wedge Z\rangle \rightsquigarrow \langle \exists\, x_1, \ldots, x_k :: x_i\ R \cdot S\ x_j \wedge Z\rangle$	
$c \notin vars(Z) \Rightarrow \langle \exists\, x_1, \ldots, c, \ldots, x_k :: x_i\ R\ c \wedge x_j\ S\ c \wedge Z\rangle \rightsquigarrow \langle \exists\, x_1, \ldots, x_k :: x_i\ R \cdot S^\circ\ x_j \wedge Z\rangle$	
$c \notin vars(Z) \Rightarrow \langle \exists\, x_1, \ldots, c, \ldots, x_k :: c\ R\ x_i \wedge c\ S\ x_j \wedge Z\rangle \rightsquigarrow \langle \exists\, x_1, \ldots, x_k :: x_i\ R^\circ \cdot S\ x_j \wedge Z\rangle$	
$c \notin vars(Z) \Rightarrow \langle \exists\, x_1, \ldots, c, \ldots, x_k :: c\ R\ x_i \wedge x_j\ S\ c \wedge Z\rangle \rightsquigarrow \langle \exists\, x_1, \ldots, x_k :: x_j\ S \cdot R\ x_i \wedge Z\rangle$	
$\langle \forall\, x_1, \ldots, c, \ldots, x_k : c\ R\ x_i : c\ S\ x_j\rangle \rightsquigarrow \langle \forall\, x_1, \ldots, x_k :: x_i\ R \backslash S\ x_j\rangle$	
$\langle \forall\, x_1, \ldots, c, \ldots, x_k : c\ R\ x_i : x_j\ S\ c\rangle \rightsquigarrow \langle \forall\, x_1, \ldots, x_k :: x_i\ R \backslash S^\circ\ x_j\rangle$	
$\langle \forall\, x_1, \ldots, c, \ldots, x_k : x_i\ R\ c : x_j\ S\ c\rangle \rightsquigarrow \langle \forall\, x_1, \ldots, x_k :: x_i\ R / S\ x_j\rangle$	
$\langle \forall\, x_1, \ldots, c, \ldots, x_k : x_i\ R\ c : c\ S\ x_j\rangle \rightsquigarrow \langle \forall\, x_1, \ldots, x_k :: x_i\ R / S^\circ\ x_j\rangle$	

Table 4.4: RL definitions (*definitions*).

The next step will consist on applying the definition of RL operators, moving towards PF version of the expression, and also resorting to CCR rules of those operators to achieve further simplifications. The RL definitions are defined in Table 4.4, which we will denote by *definitions*, and the CCR rules in Table 4.5, denoted by *CCRrules*.

Besides removing variables from the formula, the rule *definitions* also applies some definitions to reduce the number of operators that may be expected. Such is the case of the relational equality, variable equality (although in practice, was already applied in the previous step), and the implication. The definitions of the other operators (meet, join, split, complement, top and bottom) are direct applications of their definitions, although in some cases we resort to the converse to shift the variables to the desired position.

As for the composition, since we used the trading rule previously it is safe to disregard the range. When the variables to compose are in the wrong side, we resort to the converse to shift the variable to the other side. Also, note that there may exist other conjunctions inside the quantification, represented by Z, as long as the quantified variable does not occur in them, which is defined by $c \notin vars(Z)$. Finally, we also assume the possibility of nested quantifiers with empty ranges. These rules remove existentially quantified variables that occur exactly twice in a formula.

Concerning the division operators, like with the composition, we consider the different sides where the variable may occur, resorting to the converse, and assume that nested quantifiers with empty range may be interchangeable.

With these rules, *definitions*, *CCRrules* and *FOLrules*, we define a strategy for expression simplification:

$$simplify \triangleq many\,(FOLrules \oslash definitions \oslash CCRrules)$$

This strategy reduces the expression as much as possible before default translation kicks in. At this point, the expression might already be completely PF, or of the shape $\langle \forall\, x, y : x\ R\ y : x\ S\ y\rangle$, in which case the variables are ready to be dropped. If the expression is already PF, the default translation will do nothing: external variables will not introduced universal quantifications, and the remaining steps will do nothing since there are no variables. The second situation however would trigger the translation, complicating what was ready to go PF. As such, before moving to the default translation we try to drop the variables by applying the rule *dropExtVars*.

After the external universal variables are introduced, at the first step of the default translation, new simplifications may be performed. At this point, we already have the two external variables to which we must reduce all applications, which we will denote by x and y. Knowing these variables, we may remove quantified variables that occur exactly once in the formula, by taking advantage

id rules	$\top$ rules	$\bot$ rules
$R \cdot id \rightsquigarrow R$	$\overline{\top} \rightsquigarrow \bot$	$\bot \cdot \bot \rightsquigarrow \bot$
$id \cdot R \rightsquigarrow R$	$\top^\circ \rightsquigarrow \top$	$\bot^\circ \rightsquigarrow \bot$
$id^\circ \rightsquigarrow id$	$\delta(\top) \rightsquigarrow id$	$\delta(\bot) \rightsquigarrow \bot$
$\delta(id) \rightsquigarrow id$	$\rho(\top) \rightsquigarrow id$	$\rho(\bot) \rightsquigarrow \bot$
$\rho(id) \rightsquigarrow id$	$R \subseteq \top \rightsquigarrow true$	$\overline{\bot} \rightsquigarrow \top$
$x\ id\ x \rightsquigarrow true$		$\bot \subseteq R \rightsquigarrow true$
Meet rules	Join rules	Complement rules
$A \cap A \rightsquigarrow A$	$A \cup A \rightsquigarrow A$	$\overline{R \cdot S} \rightsquigarrow R^\circ \backslash \overline{S}$
$A \cap \top \rightsquigarrow A$	$A \cup \top \rightsquigarrow \top$	$\overline{R \backslash S} \rightsquigarrow R^\circ \cdot \overline{S}$
$A \cap \bot \rightsquigarrow \bot$	$A \cup \bot \rightsquigarrow A$	$\overline{R / S} \rightsquigarrow \overline{S} \cdot R^\circ$
$(A \cap B)^\circ \rightsquigarrow A^\circ \cap B^\circ$	$(A \cup B)^\circ \rightsquigarrow A^\circ \cup B^\circ$	$\overline{\overline{R}} \rightsquigarrow R$
$\overline{A \cup B} \rightsquigarrow \overline{A} \cap \overline{B}$	$\overline{A \cap B} \rightsquigarrow \overline{A} \cup \overline{B}$	$\overline{\overline{R}^\circ} \rightsquigarrow R^\circ$
	$\top \subseteq R \cup S \rightsquigarrow \overline{R} \subseteq S$	$\overline{S} \subseteq \overline{R} \rightsquigarrow R \subseteq S$
		$\top \subseteq \overline{R} \rightsquigarrow R \subseteq \bot$
		$\overline{R} \subseteq \top \rightsquigarrow \bot \subseteq R$
		$\bot \subseteq \overline{R} \rightsquigarrow R \subseteq \top$
		$\overline{R} \subseteq \bot \rightsquigarrow \top \subseteq R$
Converse rules	Division rules	Product rules
$R^{\circ\circ} \rightsquigarrow R$	$R^\circ \backslash S^\circ \rightsquigarrow R / S$	$\pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S \rightsquigarrow \langle R, S \rangle$
$(R \cdot S)^\circ \rightsquigarrow S^\circ \cdot R^\circ$	$\bot \backslash R \rightsquigarrow true$	$\langle \pi_1, \pi_2 \rangle \rightsquigarrow id$
$\top \subseteq R^\circ \rightsquigarrow \top \subseteq R$	$R \backslash \top \rightsquigarrow true$	$\pi_1 \cdot \langle R, S \rangle \rightsquigarrow R \cdot \delta(S)$
$R^\circ \subseteq \top \rightsquigarrow R \subseteq \top$	$R^\circ / S^\circ \rightsquigarrow R \backslash S$	$\pi_2 \cdot \langle R, S \rangle \rightsquigarrow S \cdot \delta(R)$
$\bot \subseteq R^\circ \rightsquigarrow \bot \subseteq R$	$\bot / R \rightsquigarrow true$	$\langle R, S \rangle^\circ \cdot \langle A, B \rangle \rightsquigarrow R^\circ \cdot A \cap S^\circ \cdot B$
$R^\circ \subseteq \bot \rightsquigarrow R \subseteq \bot$	$R / \top \rightsquigarrow true$	$\langle \top, \top \rangle \rightsquigarrow \top$
$id \subseteq R^\circ \rightsquigarrow id \subseteq R$	$\top \subseteq R \backslash \bot \rightsquigarrow \top \subseteq \overline{R}$	$\langle id, \top \rangle \cdot R \rightsquigarrow \langle R, \top \cdot R \rangle$
$R^\circ \subseteq id \rightsquigarrow R \subseteq id$	$\top \subseteq \top \backslash R \rightsquigarrow R$	$\langle \top, id \rangle \cdot R \rightsquigarrow \langle \top \cdot R, R \rangle$
	$\top \subseteq R / \bot \rightsquigarrow \top \subseteq \overline{R}$	$\langle R \cdot \pi_1, S \cdot \pi_2 \rangle \rightsquigarrow R \times S$
	$\top \subseteq \top / R \rightsquigarrow R$	$id \times id \rightsquigarrow id$
Coproduct rules		GC
$R \cdot i_1^\circ \cup S \cdot i_2^\circ \rightsquigarrow [R, S]$		$f\ function \Rightarrow R \subseteq f^\circ \cdot S \rightsquigarrow f \cdot R \subseteq S$
$[i_1, i_2]\rangle \rightsquigarrow id$		$f\ function \Rightarrow f^\circ \cdot R \subseteq S \rightsquigarrow R \subseteq f \cdot S$
$[R, S] \cdot [A, B]^\circ \rightsquigarrow R \cdot A^\circ \cup S \cdot B^\circ$		$R \subseteq S \backslash T \rightsquigarrow S \cdot R \subseteq T$
$[i_1 \cdot R, i_2 \cdot S] \rightsquigarrow R + S$		$R \subseteq S / T \rightsquigarrow R \cdot T \subseteq S$

Table 4.5: CCR transformations (*CCRrules*).

of the definition of $\top$, $\langle \exists\, k :: x\ R\ k \rangle \equiv \langle \forall\, a :: x\ R \cdot \top\ a \rangle$:

$$\begin{aligned}
\langle \exists\, x_1, \ldots, c, \ldots, x_k :: a\ R\ c \wedge Z \rangle \wedge c \notin vars(Z) &\rightsquigarrow \langle \exists\, x_1, \ldots, x_k :: x\ \top \cdot R^\circ\ a \wedge Z \rangle \\
\langle \exists\, x_1, \ldots, c, \ldots, x_k :: c\ R\ a \wedge Z \rangle \wedge c \notin vars(Z) &\rightsquigarrow \langle \exists\, x_1, \ldots, x_k :: x\ \top \cdot R\ a \wedge Z \rangle \\
\langle \forall\, x_1, \ldots, c, \ldots, x_k :: c\ R\ a \rangle &\rightsquigarrow \langle \forall\, x_1, \ldots, x_k :: x\ R \backslash \top\ a \rangle \\
\langle \forall\, x_1, \ldots, c, \ldots, x_k :: a\ R\ c \rangle &\rightsquigarrow \langle \forall\, x_1, \ldots, x_k :: x\ R / \top\ a \rangle
\end{aligned} \tag{4.19}$$

Note that since this transformation may introduce the external variable x before the *uniformIO* rule is applied, *uniformIO* will need to check if x is not already the output, in which case it will not convert it.

Application of these rules may enable the applications of the previous quantifier rules, for two occurrences. As such, we define a new version of the *simplify* rule, which includes these new rules:

$$simplify' \triangleq many\,(FOLrules \oslash definitions \oslash CCRrules \oslash once\,(4.19))$$

At this point we have done all we can to remove universal quantifications without resorting to the default method, so the default translation kicks in. We apply the *simplify′* rule after every step.

Once the expression is completely PF and an inequation is obtained, the rules defined in Table 4.5 for inequations may be applied. These rules are based on the definitions of RL operators and on GCs. Although GCs they are very powerful for expression manipulation, in our case we will only use a restricted set of rules, which may simplify our inequations. Our approach gives preference to the composition over the other operators, because composition is more rich in properties.

The new translation, the default one enhanced with the heuristic rules, is thus defined by:

$$\begin{aligned}
&simplify \triangleright (dropExtVars \oslash (insExtVars \triangleright simplify' \triangleright remUniversal \triangleright \\
&\quad simplify' \triangleright uniformIO \triangleright many\,(dropExtVars \oslash (reduceBC \oslash remQuant)))) \triangleright simplify'
\end{aligned}$$

Lets apply this translation to an example to see it in action. Consider the following expression:

$$\langle \forall\, a, a' :: a\ id\ a' \wedge a'\ S\ a \Rightarrow \langle \exists\, l, m, n : l\ R^\circ\ a' : m\ R\ l \wedge l\ S\ n \rangle \rangle$$

By applying our default translation, we obtain the following formula:

$$\top \subseteq \overline{\overline{\top \cdot (\pi_1 \cap id \cdot \pi_2) \cap \top \cdot (\pi_2 \cap S \cdot \pi_1) \cup (\top \cdot (\pi_2 \cdot \pi_1 \cdot \pi_1 \cap R^\circ \cdot \pi_2 \cdot \pi_1 \cdot \pi_1 \cdot \pi_1) \cap}}$$
$$\overline{\overline{\top \cdot (\pi_2 \cdot \pi_! \cap R \cdot \pi_2 \cdot \pi_1 \cdot \pi_1) \cap \top \cdot (\pi_2 \cdot \pi_1 \cdot \pi_1 \cap S \cdot \pi_2) \cdot \langle id, \top \rangle \cdot \langle id, \top \rangle \cdot \langle id, \top \rangle)} \cdot \langle id, \top \rangle \cdot \top}$$

On the other hand, by applying the heuristics defined in this section, we obtain:

$$\begin{aligned}
&\langle \forall\, a, a' :: a\ id\ a' \wedge a'\ S\ a \Rightarrow \langle \exists\, l, m, n : l\ R^\circ\ a' : m\ R\ l \wedge l\ S\ n \rangle \rangle \\
\Leftrightarrow &\{simplify\} \\
&\langle \forall\, a : a\ S\ a : \langle \exists\, m, n :: (a, m)\ \langle R, R \rangle \cdot S\ n \rangle \rangle \\
\Leftrightarrow &\{insExtVars\} \\
&\langle \forall\, x, y, a : a\ S\ a : \langle \exists\, m, n :: (a, m)\ \langle R, R \rangle \cdot S\ n \rangle \rangle \\
\Leftrightarrow &\{simplify'\} \\
&\langle \forall\, x, y, a : a\ S\ a : \langle \exists\, m :: x\ \top \cdot S^\circ \cdot \langle R, R \rangle^\circ\ (a, m) \rangle \rangle \\
\Leftrightarrow &\{remUniversal\} \\
&\langle \forall\, x, y :: \neg \langle \exists\, a :: a\ S\ a \wedge \neg \langle \exists\, m :: x\ \top \cdot S^\circ \cdot \langle R, R \rangle^\circ\ (a, m) \rangle \rangle \rangle \\
\Leftrightarrow &\{uniformIO\} \\
&\langle \forall\, x, y :: \neg \langle \exists\, a :: x\ \top \cdot (id \cap S)\ a \wedge \neg \langle \exists\, m :: x\ \top \cdot S^\circ \cdot \langle R, R \rangle^\circ\ (a, m) \rangle \rangle \rangle \\
\Leftrightarrow &\{remQuant\}
\end{aligned}$$

$$
\begin{aligned}
&\langle \forall\, x, y :: \neg\langle \exists\, a :: x\ \top \cdot (id \cap S)\ a \wedge \neg x\ \top \cdot S^{\circ} \cdot \langle R, R\rangle^{\circ} \cdot \langle id, \top\rangle\ a\rangle\rangle \\
\Leftrightarrow &\{reduceBC\} \\
&\langle \forall\, x, y :: \neg\langle \exists\, a :: x\ \top \cdot (id \cap S) \cap \overline{\top \cdot S^{\circ} \cdot \langle R, R\rangle^{\circ} \cdot \langle id, \top\rangle}\ a\rangle\rangle \\
\Leftrightarrow &\{remQuant\} \\
&\langle \forall\, x, y :: \neg x\ (\top \cdot (id \cap S) \cap \overline{\top \cdot S^{\circ} \cdot \langle R, R\rangle^{\circ} \cdot \langle id, \top\rangle}) \cdot \top\ y\rangle \\
\Leftrightarrow &\{reduceBC\} \\
&\langle \forall\, x, y :: x\ \overline{\top \cdot (id \cap S) \cap \overline{\top \cdot S^{\circ} \cdot \langle R, R\rangle^{\circ} \cdot \langle id, \top\rangle} \cdot \top}\ y\rangle \\
\Leftrightarrow &\{dropExtVars\} \\
&\top \subseteq \overline{\top \cdot (id \cap S) \cap \overline{\top \cdot S^{\circ} \cdot \langle R, R\rangle^{\circ} \cdot \langle id, \top\rangle} \cdot \top} \\
\Leftrightarrow &\{RLrules\} \\
&\top \cdot (id \cap S) \subseteq \top \cdot S^{\circ} \cdot (R^{\circ} \cap R^{\circ} \cdot \top)
\end{aligned}
$$

Chapter 5

Alloy to RL translation

Since we have already presented a translation from RL to CCR, a PW to PF transformation, it remains to define how the Alloy specifications are represented in RL. We start by restricting the Alloy language so a simpler subset. Next, we analyze how n-ary relations may be represented by binary relations in RL. After that, we define our type system, compatible with the types of Alloy. Finally, we define how to translate the Alloy formulas to RL. To simplify the representation and manipulation of n-ary relations, we will also define special n-ary operators for that purpose. The chapter ends with an example of this translation.

5.1 Reducing Alloy to a smaller subset

As seen in 2.3, Alloy supports a variety of notation styles. However, when dealing with an automatic translation, it is simpler to restrict the language to a smaller subset. So, the first step of the translation is to uniform certain characteristics of Alloy specifications. Through the rest of the chapter, the subset of the Alloy language we will consider is defined by the grammar in Figure 5.1. Note that the subset we consider is similar to the core defined by the authors of Alloy [Jac06], excluding the comprehension lists and the closure operators. As such, it is possible to reduce most Alloy specifications to this subset. Another characteristic we do not consider is the overloading of relations. We will now briefly explain how some reductions were realized, and what was left out.

Signature properties Some of the properties defined in the signatures, like the signature facts and relations with expressions as fields, are previously converted to independent facts, which can be trivially done. We do leave the multiplicity operators applied to the signatures and to the relations, since they will directly introduce interesting PF properties into the model.

Quantifications Both the multiplicity options for quantifications over variables and over the cardinality expressions are reduced. In particular, the multiplicity `one` is always defined as the conjunction of `some` and `lone`, and the keyword `no` as `not some`. As for the multiplicity `lone`, we allow it in the quantification of expressions, but on the variable quantifications we translate it to an equivalent definition using universal quantifications:

$$\texttt{lone}\ \texttt{x} : \texttt{R} \mid \texttt{S} \rightsquigarrow \texttt{all}\ \texttt{x}, \texttt{x}' : \texttt{R} \mid \texttt{S} \mathbin{\&\&} \texttt{S}[\texttt{x} := \texttt{x}'] \Rightarrow \texttt{x} = \texttt{x}'$$

We also assume that multiple variable declarations within a single quantification are extended to one quantifier for each variable.

We leave higher-order quantifications out of the translation, since RL has no support for those. In fact, the Alloy Analyzer also has limited support for those quantifications.

Prog : Paragraph*

Paragraph : Sig
| Fact
| Assert

Sig :
[Mult] [**abstract**] **sig** sigN [**extends** sigN] {Rel*}

Fact : **fact** {Form}

Assert : **assert** {Form}

Rel : [Mult] sigN
| [Mult] sigN → Rel

Mult : **some** | **one** | **lone**

Form : Exp ⊆ Exp
| Exp = Exp
| Form && Form
| Form || Form
| Form ⇒ Form
| Form ⇔ Form
| ! Form
| **all** varN : Exp | Form
| **some** varN : Exp | Form
| **some** Exp
| **lone** Exp

Exp : varN
| relN
| sigN
| **iden**
| **none**
| **univ**
| ∼Exp
| Exp · Exp
| Exp & Exp
| Exp + Exp
| Exp − Exp
| Exp → Exp
| Exp ◁ Exp
| Exp ▷ Exp

Figure 5.1: Alloy's restricted grammar.

Constraints In Alloy, constraints are defined in what is called paragraphs, lines of formulas which are implicitly connected by conjunction operators. For simplicity purposes, we assume that that conjunction will be explicitly defined, and thus, paragraphs become a single formula.

Predicate and function calls are also expanded before the translation. Every call is replaced by the respective expression, assigning the concrete values to the arguments. As such, it may be assumed that every variable that occurs in a expression is already quantified.

Formulas We assume that some constructs, easily defined by the other operators are previously expanded to their definition. Such is the case of let statements, if-then-else expressions and the box join operator.

The most significant limitation of our subset is that it does not allow the use of the closure and comprehension. This happens because there is no simple way to represent these operators in CCR, which is the goal of our translation.

Commands The commands are used to instruct the Analyzer on what to do, and thus, are not part of the model, and are disregarded during the translation.

5.2 Representing n-ary relations

Alloy allows relations of any arity, not only arity higher than 2, but also unary relations. However, in RL only binary relations are valid, and thus, we need a way to represent n-ary elements in that framework. Since relations are a subset of the cartesian product, a relation of type $A \leftarrow B$ can be represented by a set of type $A \times B$. As such, we will represent any Alloy relation $\mathtt{R} : A_1 \rightarrow \cdots \rightarrow A_n$ as relation $R : (A_1 \times \cdots \times A_{n-1}) \leftarrow A_n$, i.e., we leave the last field as the domain, and define the range as a product of the remaining fields, associated to the left. The order of the input/output is also reverse, due to the fact that our translation will be done in a categorical setting. Throughout the presentation, if a relation is denoted as $\mathtt{R}$, we assume it is typed in its original form, as a n-ary relation, and if it is represented by R, we assume it represents its translation to a binary relation, with the range as a product, in reverse order. Also, we will often abuse the definition of arity and refer to the binary version of the relation as n-ary, for simplicity purposes: when we say a binary relation is n-ary, we mean that its range is a product with $n-1$ elements, and the domain is a single element.

Note that although this transformation represents the same relation, some operators that are valid for relations $\mathtt{R} :: \mathtt{A_1} \rightarrow \cdots \rightarrow \mathtt{A_n}$ and $\mathtt{S} :: \mathtt{B_1} \rightarrow \cdots \rightarrow \mathtt{B_m}$ are not for the transformations $R :: (A_1 \times \cdots \times A_{n-1}) \leftarrow A_n$ and $S :: (B_1 \times \cdots \times B_{m-1}) \leftarrow B_m$. Such is the case of the Alloy

navigation: although `R · S` might be a valid composition, $R \cdot S$ in RL may not be, since the types A_n and $B_1 \times \cdots \times B_{n-1}$ do not match. In section 5.5 we will present a operator which composes relations of that type.

As for unary expressions, they may either be variable occurrences, unary constants (signatures, `univ` and `none` operators), or unary terms resulting from navigational expressions involving unary relations. As will be presented in section 5.4, we will expand the Alloy terms to its full PW representation. As such, variables will only appear in applications, and not as unary relations. Unary constants will be translated to binary relations in RL: signatures to identity relations, `univ` to $\top$ and `none` to $\bot$. In the end, our translation will not contain any unary relations, so we do not need to worry about their treatment.

5.3 Type hierarchy

The representation of Alloy specifications in a categorical setting benefits from the fact that the categories are typed. So, each Alloy signature will represent a type in the CCR. Considering the hierarchy, since sub-signatures are disjoint, super-signatures will be defined as the coproduct of their sub-signatures, if it is abstract. If it is not abstract, a new type is assumed to exist, which contains the elements which don't belong to neither sub-type. This is actually the approach taken by the authors of Alloy when formalizing its type system [EJT04].

Alloy allows the combination of any relations, even if their types are not related. Application of variables that are not a sub- nor super-type is also allowed, with only a warning being thrown. On the other hand, types in categorical RL must always match. As such, it is necessary for our type system to be sufficiently versatile to admit those possibilities, while still resulting in valid CCR expressions.

We denote by *basic types* the types defined by a signature, i.e., types which are either simple types or coproducts. The hierarchy of the basic types will be represented by a *bounded lattice*. A lattice is a partial ordered set (poset), where some elements are related by a binary relation $\trianglelefteq$, every two elements have a unique least upper bound, called *join*, and represented by $\Uparrow$, and a unique greatest lower bound, called *meet*, represented by $\Downarrow$. The order $\trianglelefteq$ is reflexive, antisymmetric and transitive, and both $\Uparrow$ and $\Downarrow$ are commutative, associative and idempotent. A bounded lattice is a lattice with a greatest and a least element.

If we see the types as a set of values, a type α is a sub-type of β if $\alpha \subseteq \beta$, and thus, that is the definition of the $\trianglelefteq$ operator. On the other hand, the meet and join operators of the lattices, $\Downarrow$ and $\Uparrow$, represent the meet and join operators of set theory, $\cap$ and $\cup$. Note that since these operators are associative, commutative and idempotent, elements of the lattice are normalized, i.e., for instance, both $A + (B + C)$ and $B + (A + C)$ represent the same element in the lattice. Also, with this definition, the greatest and least elements are represented by the *univ* constant of Alloy (a set containing all unary relations) and the *none* constant (an empty unary relation).

Lets see how this works in practice with an example. Consider Alloy declarations:

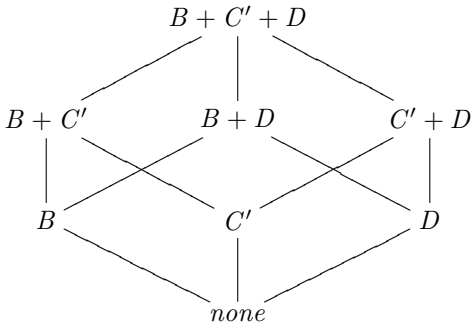
```
abstract sig A {}
sig B {}
sig C extends A {}
sig D extends C {}
```

First, we extend the definitions of the super-types:

$$C \cong C' + D$$
$$A \cong C' + D$$
$$univ \cong B + C' + D$$

Here C' represents the values of C that do not belong to its sub-type D. This structure corresponds

to the lattice:



Since the type of an applied variable may be a sub-type, super-type, or not related at all to the type of the relation, we need to convert one to the other via a common super-type, by applying combinations of injections. Since the smallest common super-type of two types α and β is given by $\alpha \Uparrow \beta$, we will always from α to β through their join.

Any type can be easily converted to one of its super-types. For any two type α and β such that $\alpha \trianglelefteq \beta$, we present the polytypic definition of an operator $\phi :: \beta \leftarrow \alpha$, that converts a value of type α into a value of type β:

$$\begin{aligned} &\phi :: \alpha \leftarrow \beta + \gamma = [\phi :: \alpha \leftarrow \beta, \phi :: \alpha \leftarrow \gamma] \\ &\phi :: \alpha + \beta \leftarrow \gamma = \begin{cases} i_1 \cdot (\phi :: \alpha \leftarrow \gamma) & \text{if } \gamma \trianglelefteq \alpha \\ i_2 \cdot (\phi :: \beta \leftarrow \gamma) & \text{if } \gamma \trianglelefteq \beta \end{cases} \\ &\phi :: \alpha \leftarrow \alpha = id \end{aligned} \tag{5.1}$$

If the sub-type is a simple type, ϕ is the combination of injections which converts it to its super-type. If the sub-type is a coproduct, eithers are previously applied to the conversion, so that each simple type in the coproduct can be treated individually. Note that if $\alpha = \beta$, $\phi :: \alpha \leftarrow \beta$ must result in id. In fact, if α is a simple type, ϕ directly results in id. If α is a coproduct $\gamma + \delta$, then, $\phi :: \gamma + \delta \leftarrow \gamma + \delta$ results in $[i_1, i_2]$, which by the reflexion law of coproducts, is equal to id.

As an example, lets calculate $\phi :: (A + B) + C \leftarrow A + C$:

$$\begin{aligned} &\phi :: (A + B) + C \leftarrow A + C \\ \Leftrightarrow&\{\ (5.1)\ \} \\ &[\phi :: (A + B) + C \leftarrow A, \phi :: (A + B) + C \leftarrow C] \\ \Leftrightarrow&\{\ (5.1)\ \} \\ &[i_1 \cdot (\phi :: (A + B) \leftarrow A), i_2 \cdot (\phi :: C \leftarrow C)] \\ \Leftrightarrow&\{\ (5.1)\ \} \\ &[i_1 \cdot i_1 \cdot id, i_2 \cdot id] \\ \Leftrightarrow&\{+\text{-Rules}\} \\ &i_1 + id \end{aligned}$$

With ϕ defined, which converts a type α to β when $\alpha \trianglelefteq \beta$, it remains to define an operation that converts any type α to any type β, where not necessarily $\alpha \trianglelefteq \beta$. Since we know that $\alpha \trianglelefteq (\alpha \Uparrow \beta) \trianglerighteq \beta$, we can use the type $\alpha \Uparrow \beta$ as an intermediate type of the conversion. As such, for any two type α and β, we define the conversion from β to α by $\psi :: \alpha \leftarrow \beta$:

$$\psi :: \alpha \leftarrow \beta = (\phi :: \alpha \Uparrow \beta \leftarrow \alpha)^\circ \cdot (\phi :: \alpha \Uparrow \beta \leftarrow \beta) \tag{5.2}$$

Note that if, for instance, $\beta \trianglelefteq \alpha$, then $\alpha = \alpha \Uparrow \beta$, and as such, ψ simplifies to $(\phi :: \alpha \Uparrow \beta)^\circ$, since $\psi :: \alpha \leftarrow \alpha = id$. In the even more particular case where $\alpha = \beta$, $\psi = id$.

Now that we have shown how to reason about basic types, we need to consider arbitrary types. For the purpose of this explanation, we will consider relations as subsets of the cartesian product, so that $\alpha \leftarrow \beta$ is represented by $\alpha \times \beta$. There is an essential difference in the way these types are represented in Alloy and the way we wish to represent them in RL. In Alloy, arbitrary types are represented as *sums of products* [EJT04], while we want to represent them as *products of sums*, so that we can represent and manipulate variables as tuples.

Given an Alloy expression $\mathtt{R} :: \alpha$, we will denote by α_k the sum of the k-th element of each product of Alloy type α, i.e., for an Alloy type $(A_1 \times \cdots \times A_n) + \cdots + (Z_1 \times \cdots \times Z_n)$, $\alpha_k = A_k + \cdots + Z_k$. This operation is valid since the typing rules of Alloy guarantee that every product term has the same number of elements [EJT04]. The full conversion of the type is defined by $\alpha_1 \times \cdots \times \alpha_n = (A_1 + \cdots + Z_1) \times \cdots \times (A_n + \cdots + Z_n)$. Note that these two types are not isomorphic: the later is an over-approximation of the original Alloy type. This will not pose a problem, since the translation will deal with the redundant types by using ψ operations to filter out the invalid values. We also define and operator #, which for every Alloy type return its arity, for example, $\#\alpha = n$.

As an example, in Alloy, the union of two relations $\mathtt{R} :: A \leftarrow B$ and $\mathtt{S} : C \leftarrow D$ has type $(A \leftarrow B) + (C \leftarrow D)$. In RL however, it will be translated to an expression of type $(A + C) \leftarrow (B + D)$. This type is an over-approximation of the first, since it is isomorphic to $(A \leftarrow B) + (A \leftarrow D) + (C \leftarrow B) + (C \leftarrow D)$. However, these extra cases will be filtered out by the ψ operations, since they will result is mismatching injections.

5.4 Formula translation

The main goal of an Alloy to RL translation is the representation of its formulas. In our case, since we have previously restricted the Alloy language, this means translating assertions and facts. Although they will serve different purposes, their translation, which we will now present, is done the same way.

Translating Alloy expressions to RL would be fairly direct if not for the presence of n-ary relations. In particular, the possibility of combining unary and higher than binary relations in navigational expression makes it impossible to directly translate them to RL. For instance, for two relations $\mathtt{R}$ and $\mathtt{S}$ with arity i and j respectively, and $\mathtt{x}$ a variable, it is not clear how to translate $\mathtt{R} \cdot \mathtt{x} \cdot \mathtt{S}$ directly to CCR. A solution, proposed in [FPA04], is to translate all unary relations to correflexives. However, we want to fully remove variables from the expressions, and not only "hiding" them into correflexives, even though it results in PF expressions. Since directly translating the operators is not an option, we will fully expand them to PW during the translation. In practice, we are translating the operators to their semantic in RL. Once fully expanded, we apply the transformation defined in section 4.6, which will translate it to CCR, resulting in a PF expression.

Alloy formulas are essentially RL, so it is easy to define their semantics:

$$
\begin{aligned}
&[\![!\mathtt{R}]\!] \equiv \neg[\![\mathtt{R}]\!] \\
&[\![\mathtt{R}\&\&\mathtt{S}]\!] \equiv [\![\mathtt{R}]\!] \wedge [\![\mathtt{S}]\!] \\
&[\![\mathtt{R} \mid\mid \mathtt{S}]\!] \equiv [\![\mathtt{R}]\!] \vee [\![\mathtt{S}]\!] \\
&[\![\mathtt{R} \Rightarrow \mathtt{S}]\!] \equiv [\![\mathtt{R}]\!] \Rightarrow [\![\mathtt{S}]\!] \\
&[\![\mathtt{R} \Leftrightarrow \mathtt{S}]\!] \equiv [\![\mathtt{R}]\!] \Leftrightarrow [\![\mathtt{S}]\!] \\
&[\![\mathtt{All}\,\mathtt{x} : (\mathtt{X} :: A) \mid \mathtt{S}]\!] \equiv \langle \forall\, x \in A : [\![\mathtt{x} \subseteq \mathtt{X}]\!] : [\![\mathtt{S}]\!] \rangle \qquad (5.3) \\
&[\![\mathtt{Some}\,\mathtt{x} : (\mathtt{X} :: A) \mid \mathtt{S}]\!] \equiv \langle \exists\, x \in A : [\![\mathtt{x} \subseteq \mathtt{X}]\!] : [\![\mathtt{S}]\!] \rangle \\
&[\![\mathtt{Some}\,(\mathtt{X} :: A)]\!] \equiv \langle \exists\, x_1 \in A_1, \ldots, x_{\#A} \in A_{\#A} :: [\![(x_1, \ldots, x_{\#A}) \in \mathtt{X}]\!] \rangle \\
&[\![\mathtt{X} = \mathtt{Y}]\!] \equiv [\![\mathtt{X} \subseteq \mathtt{Y}]\!] \wedge [\![\mathtt{Y} \subseteq \mathtt{X}]\!] \\
&[\![(\mathtt{X} :: A) \subseteq (\mathtt{Y} :: B)]\!] \equiv \\
&\qquad \langle \forall\, x_1 \in (A_1 \Uparrow B_1), \ldots, x_{\#A} \in (A_{\#A} \Uparrow B_{\#B}) : [\![(x_1, \ldots, x_{\#A}) \in \mathtt{X}]\!] : [\![(x_1, \ldots, x_{\#A}) \in \mathtt{Y}]\!] \rangle
\end{aligned}
$$

where R and S are formulas, and X and Y are expressions. The translation of the boolean operators is direct. As for the quantifiers, note that a range X of a quantified variable x in Alloy is translated to $\mathtt{x} \subseteq \mathtt{X}$, which is its actual meaning. In our restricted Alloy, every constraint eventually ends with a $\subseteq$ operator. Like we said, we will expand the definitions of the operators to PW, and thus the insertion of the universal quantifications when translating the $\subseteq$ operator. Those variables must be passed through the expressions so that their PW definition is applied, so, by abusing the notation, we use the expression $x \in \mathtt{X}$ to state a that an Alloy expression X contains the variable x. As such, the semantics of the Alloy expression in PW is defined by:

$$
\begin{aligned}
&[\![(x :: A) \in (\mathtt{v} :: B)]\!] \equiv x\,(\phi :: A \leftarrow B)\,v \\
&[\![(x :: A) \in \mathtt{S}]\!] \equiv x\,(\psi :: A \leftarrow S) \cdot (\psi :: S \leftarrow A)\,x \\
&[\![(x_1 :: A_1, \ldots, x_n :: A_n) \in (\mathtt{R} :: B)]\!] \equiv \\
&\qquad (x_1, \ldots, x_{n-1})\,((\psi :: A_1 \leftarrow B_1) \times \cdots \times (\psi :: A_{n-1} \leftarrow B_{n-1})) \cdot R \cdot (\psi :: B_n \leftarrow A_n)\,x_n \\
&[\![(x_1 :: A, x_2 :: B) \in \mathtt{iden}]\!] \equiv x_1\,(\psi :: A \leftarrow \mathit{univ}) \cdot (\psi :: \mathit{univ} \leftarrow B)\,x_2 \\
&[\![(x :: A) \in \mathtt{univ}]\!] \equiv x\,(\psi :: A \leftarrow \mathit{univ}) \cdot \top_{\mathit{univ} \leftarrow \mathit{univ}} \cdot (\psi :: \mathit{univ} \leftarrow B)\,x \\
&[\![(x :: A) \in \mathtt{none}]\!] \equiv \mathit{false} \\
&[\![(x_1, \ldots, x_n) \in \mathtt{X} + \mathtt{Y}]\!] \equiv [\![(x_1, \ldots, x_n) \in \mathtt{X}]\!] \vee [\![(x_1, \ldots, x_n) \in \mathtt{Y}]\!] \\
&[\![(x_1, \ldots, x_n) \in \mathtt{X} \,\&\, \mathtt{Y}]\!] \equiv [\![(x_1, \ldots, x_n) \in \mathtt{X}]\!] \wedge [\![(x_1, \ldots, x_n) \in \mathtt{Y}]\!] \\
&[\![(x_1, \ldots, x_n) \in \mathtt{X} - \mathtt{Y}]\!] \equiv [\![(x_1, \ldots, x_n) \in \mathtt{X}]\!] \wedge \neg [\![(x_1, \ldots, x_n) \in \mathtt{Y}]\!] \\
&[\![(x_1, x_2) \in \sim \mathtt{X})]\!] \equiv [\![(x_2, x_1) \in \mathtt{X}]\!] \\
&[\![(x_1, \ldots, x_n) \in (\mathtt{X} :: A) \cdot (\mathtt{Y} :: B) \equiv \\
&\qquad \langle \exists\, k \in A_{\#A} \Uparrow B_1 :: [\![(x_1, \ldots, x_{\#A-1}, k) \in \mathtt{X}]\!] \wedge [\![(k, x_{\#A}, \ldots, x_n) \in \mathtt{Y}]\!] \rangle \\
&[\![(x_1, \ldots, x_n) \in (\mathtt{X} :: A) \to (\mathtt{Y} :: B) \equiv [\![(x_1, \ldots, x_{\#A}) \in X]\!] \wedge [\![(x_{\#A+1}, \ldots, x_n) \in Y]\!] \\
&[\![(x_1, \ldots, x_n) \in \mathtt{X} <: \mathtt{Y}]\!] \equiv [\![x_1 \in \mathtt{X}]\!] \Rightarrow [\![(x_1, \ldots, x_n) \in \mathtt{Y}]\!] \\
&[\![(x_1, \ldots, x_n) \in \mathtt{X} :> \mathtt{Y}]\!] \equiv [\![x_n \in \mathtt{Y}]\!] \Rightarrow [\![(x_1, \ldots, x_n) \in \mathtt{X}]\!]
\end{aligned}
\tag{5.4}
$$

where X and Y are expressions, R is a relation, v a variable and S a signature. Note that since all operators are extended to PW, type conversion, using ψ and ϕ, is only applied at the variable application level. When the relations are translated, we perform a product of type transformations, to convert the type of each element of the output tuple. In the translation of signature, ψ will result in $\bot$ if the type of the variable is neither a sub- nor super-type of the signature.

The RL operators meet, join, difference and converse are directly translated. As for the navigation, the types might not match, and so once again we must calculate their least super-type. In case one of the terms of the navigation is a variable, the newly inserted existential quantification will later be cut by the one-point rule.

The cartesian product of two expressions just states that in order to belong to it, the variables must belong to those expressions. Domain and range restrictions of an expression state that in order to belong to the expression, the last or first element of the tuple, respectively, must belong to the restriction.

Lets see the efect of this translation with an example. Consider the Alloy formula:

$$\mathtt{all\ a} : \mathtt{R.B} \mid \mathtt{R} \subseteq \mathtt{S} \cdot \mathtt{a} \cdot \mathtt{R}$$

where $\mathtt{R} : A \to B$ and $\mathtt{S} : A \to A \to A + B$. Assume that in the following calculus, the type calculation in done implicitly, only appearing when the types do not match. By applying (5.3) to the formula, we get:

$$
\begin{aligned}
&[\![\mathtt{all\ a} : \mathtt{R.B} \mid \mathtt{R} \subseteq \mathtt{S} \cdot \mathtt{a} \cdot \mathtt{R}]\!] \\
\Leftrightarrow &\{\ (5.3)\ \} \\
&\langle \forall\, a \in A : [\![\mathtt{a} \subseteq \mathtt{R} \cdot \mathtt{B}]\!] : [\![\mathtt{R} \subseteq \mathtt{S} \cdot \mathtt{a} \cdot \mathtt{R}]\!] \rangle
\end{aligned}
$$

We will expand the transformations of the $\subseteq$ operators individually, for simplicity purposes:

$$
\begin{array}{l}
\quad [\![\mathtt{a} \subseteq \mathtt{R.B}]\!] \\
\Leftrightarrow\{\ (5.3)\ \} \\
\quad \langle \forall\, x \in A : [\![x \in \mathtt{a}]\!] : [\![x \in \mathtt{R} \cdot \mathtt{B}]\!] \rangle \\
\Leftrightarrow\{\ (5.4)\ \} \\
\quad \langle \forall\, x \in A : x = a : \langle \exists\, k :: [\![(x,k) \in \mathtt{R}]\!] \wedge [\![k \in \mathtt{B}]\!] \rangle \rangle \\
\Leftrightarrow\{\text{One-point, } (5.4)\ \} \\
\quad \langle \exists\, k \in B :: k\ R\ a \wedge k\ id_B\ k \rangle \\
\Leftrightarrow\{\text{One-point }\} \\
\quad \langle \exists\, k \in B :: k\ R\ a \rangle
\end{array}
$$

$$
\begin{array}{l}
\quad [\![\mathtt{R} \subseteq \mathtt{S} \cdot \mathtt{a} \cdot \mathtt{R}]\!] \\
\Leftrightarrow\{\ (5.3)\ \} \\
\quad \langle \forall\, x_1 \in A, x_2 \in B : [\![(x_1,x_2) \in \mathtt{R}]\!] : [\![(x_1,x_2) \in \mathtt{S} \cdot \mathtt{a} \cdot \mathtt{R}]\!] \rangle \\
\Leftrightarrow\{\ (5.4)\ \} \\
\quad \langle \forall\, x_1 \in A, x_2 \in B : x_2\ R\ x_1 : \langle \exists\, k \in A :: [\![(x_1,k) \in \mathtt{S} \cdot \mathtt{a}]\!] \wedge [\![(k,x_2) \in \mathtt{R}]\!] \rangle \rangle \\
\Leftrightarrow\{\ (5.4)\ \} \\
\quad \langle \forall\, x_1 \in A, x_2 \in B : x_2\ R\ x_1 : \langle \exists\, k \in A, l \in A+B :: [\![(x_1,k,l) \in \mathtt{S}]\!] \wedge [\![l \in \mathtt{a}]\!] \wedge x_2\ R\ k \rangle \rangle \\
\Leftrightarrow\{\ (5.4)\ \} \\
\quad \langle \forall\, x_1 \in A, x_2 \in B : x_2\ R\ x_1 : \langle \exists\, k \in A, l \in A+B :: (l,k)\ S\ x_1 \wedge l\ i_2\ a \wedge x_2\ R\ k \rangle \rangle
\end{array}
$$

Going back to the full formula, omitting the types, we obtain:

$$\langle \forall\, a : \langle \exists\, k :: k\ R\ a \rangle : \langle \forall\, x_1, x_2 : x_2\ R\ x_1 : \langle \exists\, k, l :: (l,k)\ S\ x_1 \wedge \wedge l\ i_2\ a \wedge x_2\ R\ k \rangle \rangle \rangle$$

5.5 N-ary operators

The Alloy translation presented in the previous section completely expands the navigation operators from Alloy. We chose to take that approach because we want to fully remove the variables from the formulas, and to do so with variables that occur inside navigations, expanding the formula to PW is needed. Once we have RL formulas, the automatic translation presented in section 4.4 could translate them to fully PF formulas. However, dealing with n-ary relations directly with that translation would result in very complex expressions, since it does not include any heuristics to deal with n-ary relations. As such, we define two new operators, the *n-ary composition* and a *rotate* operator. With these, not only the resulting expression is simpler, but we will also be able to keep part of the meaning of the original navigational expression, as in many cases, the n-ary composition will directly replace the navigation operator. The n-ary composition definition is actually similar to the one defined in [FPA04], although adapted for different settings.

For two relations $R : X_1 \times \cdots \times X_n \leftarrow A$ and $S : A \times Y_1 \times \cdots \times Y_{m-1} \leftarrow Y_m$, with any arity $n+1$ and $m+1$ respectively, the n-ary composition will be denoted by $\bullet^m$ and the result has type $R \bullet^m S : ((X_1 \times \cdots \times X_n) \times Y_1) \times \cdots \times Y_{m-1} \leftarrow Y_m$, defined by:

$$R \bullet^n S = \begin{cases} R \cdot S & \text{if } n = 1 \\ (R \times id) \bullet^{n-1} S & \text{if } n > 1 \end{cases} \tag{5.5}$$

All the products are assumed to associate to the left. Also, note that the definition depends only on the arity of the second relation, S. The behavior of the operator is similar to the Alloy navigation operator: in Alloy, for two relations $\mathtt{R} : X_1 \leftarrow \cdots \leftarrow X_n \leftarrow A$ and $\mathtt{S} : A \leftarrow Y_1 \leftarrow \cdots \leftarrow Y_m$, $\mathtt{R} \cdot \mathtt{S} : X_1 \leftarrow \cdots \leftarrow X_n \leftarrow Y_1 \leftarrow \cdots \leftarrow Y_m$.

For instance, for $R : E \times D \leftarrow C$ and $S : C \times B \leftarrow A$, $R \bullet^2 S$ has type $E \times D \times B \leftarrow A$ and its point-free meaning is:

$$
\begin{aligned}
& ((e, d), b)\ R \bullet^2 S\ a \\
\Leftrightarrow & \{\text{Def-}\bullet^n\} \\
& ((e, d), b)\ (R \times id) \cdot S\ a \\
\Leftrightarrow & \{\text{Def-Composition}\} \\
& \langle \exists\, k, l :: ((e, d), b)\ (R \times id)\ (k, l) \wedge (k, l)\ S\ a \rangle \\
\Leftrightarrow & \{\text{Def-}\times\} \\
& \langle \exists\, k, l :: (e, d)\ R\ k \wedge b\ id\ l \wedge (k, l)\ S\ a \rangle \\
\Leftrightarrow & \{\text{One-point Existential}\} \\
& \langle \exists\, k :: (e, d)\ R\ k \wedge (k, b)\ S\ a \rangle
\end{aligned}
$$

The n-ary composition benefits from a kind of associativity, has the identity relation as unit, and distributes over join, i.e., for relations R, S and T, with arity $n+1$, $m+1$ and $o+1$ respectively:

$$(R \bullet^m S) \bullet^o T \Leftrightarrow R \bullet^{m+o-1} (S \bullet^o T) \tag{5.6}$$

$$id \bullet^n R \Leftrightarrow R \Leftrightarrow R \bullet^1 id \tag{5.7}$$

$$R \bullet^m (S \cup T) \Leftrightarrow R \bullet^m S \cup R \bullet^m T \tag{5.8}$$

$$(R \cup S) \bullet^o T \Leftrightarrow R \bullet^o T \cup S \bullet^m T \tag{5.9}$$

Lets begin by proving (5.6) inductively:

$$
\begin{aligned}
& (R \bullet^m S) \bullet^n T \Leftrightarrow R \bullet^{m+n-1} (S \bullet^n T) \\
& m = 1 \\
& \quad (R \cdot S) \bullet^n T \Leftrightarrow R \bullet^n (S \bullet^n T) \\
& \quad n = 1 \\
& \quad\quad (R \cdot S) \cdot T \Leftrightarrow R \cdot (S \cdot T) \\
& \quad n > 1 \\
& \quad\quad (R \cdot S) \bullet^n T \\
& \quad\quad \Leftrightarrow \{\text{Def-}\bullet^n\} \\
& \quad\quad (R \cdot S) \times id) \bullet^{n-1} T \\
& \quad\quad \Leftrightarrow \{\times \text{ Rules}\} \\
& \quad\quad ((R \times id) \cdot (S \times id)) \bullet^{n-1} T \\
& \quad\quad \Leftrightarrow \{\text{Induction hypothesis (n)}\} \\
& \quad\quad (R \times id) \bullet^{n-1} ((S \times id) \bullet^{n-1} T) \Leftrightarrow R \bullet^n (S \bullet^n T) \\
& m > 1 \\
& \quad (R \bullet^m S) \bullet^n T \\
& \quad \Leftrightarrow \{\text{Def-}\bullet^n\} \\
& \quad (R \times id) \bullet^{m-1} S) \bullet^n T \\
& \quad \Leftrightarrow \{\text{Induction hypothesis (m)}\} \\
& \quad (R \times id) \bullet^{m+n-2} (S \bullet^n T) \\
& \quad \Leftrightarrow \{\text{Def-}\bullet^n\} \\
& \quad R \bullet^{m+n-1} (S \bullet^n T)
\end{aligned}
$$

Proving that identity is its right identity is trivial, since, by definition, $R \bullet_2 id \Leftrightarrow R \cdot id \Leftrightarrow id$. Now, lets prove that the identity is also its left unit:

$$
\begin{array}{ll}
n = 1 & \\
\quad & id \bullet^1 R \Leftrightarrow id \cdot R \Leftrightarrow id \\
n > 1 & \\
\quad & id \bullet^n R \\
\Leftrightarrow & \{\text{Def-}\bullet^n\} \\
\quad & (id \times id) \bullet^{n-1} R \\
\Leftrightarrow & \{\text{Induction hypothesis}\} \\
\quad & R
\end{array}
$$

The proof of the distributivity over join is also easily proven by induction, since composition distributes over join.

Our n-ary composition only applies when the variable to compose is on the left-most position of the output. As such, we need define an operator which rotates a tuple, which we will denote by $^{\rightarrow}$. For a relation R, it is defined by:

$$\overrightarrow{R :: \alpha} = \langle R^\circ, X_1^{\#\alpha-1}, \ldots, X_{\#\alpha-2}^{\#\alpha-1} \rangle \cdot X_{\#\alpha-1}^{\#\alpha-1\,\circ} \tag{5.10}$$

where $\langle R, \ldots, S \rangle$ represents multiple splits associated to the left, i.e. $\langle \langle R, \ldots \rangle, S \rangle$, and X_i^n is the selection operator defined in (4.2).

Lets consider a ternary relation $R : A \times B \leftarrow C$ and check the point-wise meaning of $\overrightarrow{R}$:

$$
\begin{array}{ll}
 & (a, b)\ \overrightarrow{R}\ c \\
\Leftrightarrow & \{\text{Def-}\overrightarrow{R}\} \\
 & (a, b)\ \langle R^\circ, X_1^2 \rangle \cdot X_2^{2\,\circ}\ c \\
\Leftrightarrow & \{\text{Def-Composition, Def-}X_i^m\} \\
 & \langle \exists\, k, l :: (a, b)\ \langle R^\circ, \pi_1 \rangle\ (k, l) \wedge (k, l)\ \pi_2^\circ\ c \rangle \\
\Leftrightarrow & \{\text{Def-Split,Def-}\pi\} \\
 & \langle \exists\, k, l :: a\ R^\circ\ (k, l) \wedge b\ \pi_1\ (k, l) \wedge l = c \rangle \\
\Leftrightarrow & \{\text{Eindhoven: Existential One-point, Def-}\pi\} \\
 & \langle \exists\, k :: a\ R^\circ\ (k, c) \wedge b = k \rangle \\
\Leftrightarrow & \{\text{Eindhoven: Existential One-point, Def-Converse}\} \\
 & (b, c)\ R\ a
\end{array}
$$

We will denote by $\overrightarrow{R}^{k}$ the application of the rotate k times. For R with arity n, if $k = n$ then the variables fully rotate, and thus $\overrightarrow{R}^{n} = R$. Also, applying the rotate to id, $\top$ and $\bot$ does not change them, only their type, and it distributes over join and meet. It is also monotonic, i.e.:

$$R \subseteq S \equiv \overrightarrow{R} \subseteq \overrightarrow{S} \tag{5.11}$$

The n-ary composition and converse also relate in a particular way, defined by the following property:

$$\overrightarrow{R \bullet^m S}^{m-1} \equiv \overrightarrow{S}^{m-1} \bullet^n \overrightarrow{R} \tag{5.12}$$

We these new operators, we can define rules to remove existentially quantified variable inside

		$\overrightarrow{R} \subseteq \overrightarrow{S} \rightsquigarrow R \subseteq S$
	$\overrightarrow{R}^{n} \rightsquigarrow R$	$\overrightarrow{R} \subseteq id \rightsquigarrow R \subseteq id$
$id \bullet R \rightsquigarrow R$	$\overrightarrow{\bot} \rightsquigarrow \bot$	$id \subseteq \overrightarrow{R} \rightsquigarrow id \subseteq R$
$R \bullet id \rightsquigarrow R$	$\overrightarrow{id} \rightsquigarrow id$	$\overrightarrow{R} \subseteq \top \rightsquigarrow R \subseteq \top$
$\overrightarrow{R}^{m-1} \bullet^{m} \overrightarrow{S}^{m-1} \rightsquigarrow \overrightarrow{S} \bullet^{n} \overrightarrow{R}$	$\overrightarrow{\top} \rightsquigarrow \top$	$\top \subseteq \overrightarrow{R} \rightsquigarrow \top \subseteq R$
		$\overrightarrow{R} \subseteq \bot \rightsquigarrow R \subseteq \bot$
		$\bot \subseteq \overrightarrow{R} \rightsquigarrow \bot \subseteq R$

Table 5.1: Simplification rules for the n-ary operators

tuples:

$$\langle \exists\, c :: (x_1, \ldots, x_{n-1})\, R\, c \wedge (y_1, \ldots, y_{i-1}, c, y_{i+1}, \ldots, y_{m-1})\, S\, y_m \rangle \rightsquigarrow$$
$$(x_1, \ldots, x_{n-1}, y_{i+1}, \ldots, y_{i-2})\, (R \bullet^{n} \overrightarrow{S}^{m-i+1})\, y_{i-1}$$
$$\langle \exists\, c :: (x_1, \ldots, x_{n-1})\, R\, c \wedge (y, \ldots, y_{m-1})\, S\, c \rangle \rightsquigarrow$$
$$(x_1, \ldots, x_{n-1}, y_1, \ldots, y_{m-2})\, (R \bullet^{n} \overrightarrow{S})\, y_{m-1}$$
$$\langle \exists\, c :: (x_1, \ldots, x_{j-1}, c, x_{j+1}, \ldots, x_{n-1})\, R\, x_n \wedge (y_1, \ldots, y_{i-1}, c, y_{i+1}, \ldots, y_{m-1})\, S\, y_m \rangle \rightsquigarrow$$
$$(x_{j+1}, \ldots, x_{j-1}, y_{i+1}, \ldots, y_{i-2})\, (\overrightarrow{R}^{n-j+1} \bullet^{n} \overrightarrow{S}^{m-i+1})\, y_{i-1}$$

These rules are added as new choices to the rule *definition*, defined in Table 4.4, to improve our heuristic simplifications. Although omitted for simplicity purposes, like the composition rules defined in Table 4.4 it allows the quantification to be in nested quantifications, as long as they all have empty ranges. Also, the applications may be inside other conjunctions, as long as c does not occur in them.

Since these operators may now appear on our formulas, we also add to the rule *CCRrules*, defined in Table 4.5, the rules induced by the properties of these operators, which we present in Table 5.1.

5.6 Signature and Relation multiplicity

We chose not to redefine the signature multiplicity as separate facts when we restricted the language, because this way we are able to directly retrieve interesting PF properties from them.

The default multiplicity, `set`, does not induce any property. The `some` multiplicity states that the type must not be empty. This is defined by the following inequation, for a type A:

$$\top_{univ \leftarrow univ} \subseteq \top_{univ \leftarrow A} \cdot \top_{A \leftarrow univ}$$

If A is a empty type, $\top_{univ \leftarrow 0} \cdot \top_{0 \leftarrow univ} = \bot_{univ \leftarrow univ}$, and the property fails. The `lone` multiplicity states that a type must have exactly one element. For a type A, that property is defined by:

$$\top_{A \leftarrow A} \subseteq id_A$$

The proof is simple:

$$\top_{A \leftarrow A} \subseteq id_A \Leftrightarrow \langle \forall\, x, y \in A :: x = y \rangle$$

Finally, multiplicity `one` is defined by the conjunction of `some` and `lone`. The properties induced by signature multiplicity are resumed in Table 5.2.

Sig A	Property
`set`	$true$
`some`	$\top_{univ \leftarrow univ} \subseteq \top_{univ \leftarrow A} \cdot \top_{A \leftarrow univ}$
`lone`	$\top_{A \leftarrow A} \subseteq id_A$
`one`	$\top_{univ \leftarrow univ} \subseteq \top_{univ \leftarrow A} \cdot \top_{A \leftarrow univ} \wedge \top_{A \leftarrow A} \subseteq id_A$

Table 5.2: Properties induced by the multiplicity of signatures in Alloy.

`A some` → `B`	`A` → `lone B`	`A` → `some B`	`A lone` → `B`
surjective	simple	entire	injective

`A some` → `lone B`	`A` → `one B`	`A lone` → `some B`
abstraction	function	representation

`A some` → `one B`	`A lone` → `one B`
surjection	injection

`A one` → `one B`
bijection

Table 5.3: Most common classification induced by the multiplicity of binary relations in Alloy.

Relation declarations in Alloy may also define properties by applying the multiplicity keywords `some`, `lone` and `one`. Fields with no multiplicity defined have implicitly the multiplicity `set`, which is a neutral operator, not inducing any properties. Also, note that the first field of every relation, which has the type of the signature where the relation is declared, has always multiplicity `set`.

Relation multiplicity induces in the n-ary relations a taxonomy similar to the one presented for binary relations in Figure 3.1. Regarding the binary relations, the multiplicity of the range and domain directly define the class of the relation, which is resumed in Table 5.3. Although only the most common classes are defined, using the multiplicities it is possible to obtain any combination of the four basic properties.

However, when moving to n-ary relations, there does not exist the dichotomy range/domain, so defining the properties is not so direct. Note that already when considering binary relations, one can consider each field separately: in a `A lone` → `one B` relation, one can define it as a injection, if considering it as whole, or consider first the `one` at the range, which defines it as a function, and then the `lone` at the domain, defining it as injective. That is how we analyze the multiplicity of n-ary relations: we define properties for each field at a time.

Consider, for instance, a n-ary relation, whose i-th field is marked as `lone`. That property will be defined by:

$$(\overrightarrow{R}^{n-i})^\circ \cdot \overrightarrow{R}^{n-i} \subseteq id$$

Note this expression defines a relation as simple. Lets convert it to PW to better understand its meaning. Assuming $(x_1, \ldots, x_n)_i$ contains all the variables from x_1 to x_n *except* x_i, the property induced by that field is defined by:

$$\begin{aligned}
&(\overrightarrow{R}^{n-i})^\circ \cdot \overrightarrow{R}^{n-i} \subseteq id \\
\Leftrightarrow &\{\text{Insert Points}\} \\
&\langle \forall\, x_1, \ldots, x_n, x'_1, \ldots, x'_n : \\
&\quad (x_{i+1}, \ldots, x_{i-1})_i \,(\overrightarrow{R}^{n-i})^\circ \cdot \overrightarrow{R}^{n-i}\, (x'_{i+1}, \ldots, x'_{i-1})_i : (x_{i+1}, \ldots, x_{i-1})_i = (x'_{i+1}, \ldots, x'_{i-1})_i \rangle \\
\Leftrightarrow &\{\text{Def-Composition}\} \\
&\langle \forall\, x_1, \ldots, x_n, x'_1, \ldots, x'_n : \\
&\quad \langle \exists\, x_i :: x_i\ \overrightarrow{R}^{n-i}\ (x_{i+1}, \ldots, x_{i-1})_i \wedge x_i\ \overrightarrow{R}^{n-i}\ (x'_{i+1}, \ldots, x'_{i-1})_i \rangle : (x_{i+1}, \ldots, x_{i-1})_i = (x'_{i+1}, \ldots, x'_{i-1})_i \rangle
\end{aligned}$$

i-th field	Property
`set`	$true$
`some`	$id \subseteq \overrightarrow{R}^{n-i} \cdot (\overrightarrow{R}^{n-i})^\circ$
`lone`	$(\overrightarrow{R}^{n-i})^\circ \cdot \overrightarrow{R}^{n-i} \subseteq id$
`one`	$id \subseteq \overrightarrow{R}^{n-i} \cdot (\overrightarrow{R}^{n-i})^\circ \wedge (\overrightarrow{R}^{n-i})^\circ \cdot \overrightarrow{R}^{n-i} \subseteq id$

Table 5.4: Properties induced by the multiplicity of n-ary relations in Alloy.

$$
\begin{aligned}
&\Leftrightarrow\{\text{Def-}\overrightarrow{R}^{k}\}\\
&\quad\langle \forall\, x_1, \ldots, x_n, x'_1, \ldots, x'_n :\\
&\qquad\langle \exists\, x_i :: (x_1, \ldots, x_i, \ldots, x_{n-1})\, R\, x_n \wedge (x'_1, \ldots, x_i, \ldots, x'_{n-1})\, R\, x'_n \rangle : (x_1, \ldots, x_n)_i = (x'_1, \ldots, x'_n)_i \rangle
\end{aligned}
$$

So, in practice, we rotate the variable we are dealing with to the input of the relation, and then remove it by applying a composition, resulting in the definition of a simple relation. Note however that with this method, since we choose to rotate the variable to the domain, the property induced by `lone` will always classify the rotated relation as simple, and never as injective. A particular case arises when dealing with binary relations: since rotating the relation is simply the converse, the definition of an injective relation will result when `lone` appears at the range.

The reasoning about the `some` multiplicity is similar. For a n-ary relation, whose i-th field is marked as `some`, the PF property is defined by:

$$id \subseteq \overrightarrow{R}^{n-i} \cdot (\overrightarrow{R}^{n-i})^\circ$$

We rotate the n-ary relation, and define it as surjective. As for the `one` multiplicity, it is defined by the conjunction of the properties induced by `lone` and `some`.

The translation of the multiplicity of the relations, which during the translation is added to our environment, is resumed in Table 5.4

Chapter 6

Implementation

The translations we defined in chapters 4 and 5 we implemented in a tool that transforms Alloy models into CCR inequations. This tool is divided in two parts: one to transform RL formulas into CCR, and one to parse the Alloy specifications into RL.

6.1 RL rewriter

In order to implement the translation from RL to CCR, defined in chapter 4, a generic RL rewriter was implemented. This system was implemented in the functional language Haskell as a strategic rewriting system, which allows us to rewrite expressions by defining rules and strategies. As such, once the needed RL rules were defined, we were able to define not only the translation from PW to PF, with several variants, but also its reverse, from PF to PW.

6.1.1 Datatypes

The main functionality of our tool is to transform RL expressions, so we will start by defining their types. The type used for the expressions includes the boolean, RL and categorical operators, and is recursively defined by:

```
data Exp = And Exp Exp | Or Exp Exp | Not Exp | Imply Exp Exp | Tru | Fals |
           Var String Type | Pair Exp Exp | App Exp Exp Exp | Equal | Const String Type |
           Forall String Type Exp Exp | Exists String Type Exp Exp |
           Rel String Type | In Exp Exp | RelEq Exp Exp | Comp Exp Exp | Trans Exp |
           Inter Exp Exp | Union Exp Exp | Compl Exp | Top Type | Bot Type | Id Type |
           Divr Exp Exp | Divl Exp Exp | Dom Exp | Ran Exp | ConstF String Type |
           Pi1 Type | Pi2 Type | Split Exp Exp | Prod Exp Exp |
           In1 Type | In2 Type | Either Exp Exp | Coprod Exp Exp |
           NComp Exp Exp | NTrans Exp
```

The lines define, in order, the logical operators, variables and applications, quantifiers, relational operators, product operators, co-product operators, and the n-ary operators.

Other than the expressions, the type of the terms is defined by:

```
data Type = Type String | TProd Type Type | TArrow Type Type | TCoprod Type Type
```

A function `getType` is also defined, which for any expression `Exp` returns its type.

6.1.2 Strategic Rewriting

The translations were defined using strategic rewriting techniques, which provide an effective method for rewriting terms. The process consists on defining basic rules, which may then be

combined using strategic combinators (see section 4.2). We will assume that the reader has a some basic notions about *monads* [Wad92], and their use in Haskell.

A rule is an operation that either transforms a term, if defined for its shape, or fail. In our system, rules are applied to expressions, and have the type:

```
type Rule = Exp → StateT ExpState Maybe Exp
```

The possibility of a failed transformation is introduced by the *maybe* monad and the *state* monad carries the state of the transformation. The state contains where the environment of the model, the auxiliary facts that maybe needed for rule applications, and also more technical data, such as the trace of the transformation, so that they can be presented to the user in the end, the number of occurrences of each variable, and the two external "abstract" universal quantifications.

We will now define the rule combinators. The first one is the *sequence* combinator, which given two rules, performs the second on the output if the first, if it is successful:

```
(▷) :: Rule → Rule → Rule
(f ▷ g) a = do b ← f a
               c ← g b
               return c
```

We also define the *choice* of two rules, which performs the first, and tries the second if it fails, and the *many* combinator, which performs a rule repetitively until it fails. These are defined by:

```
(⊘) :: Rule → Rule → Rule
(f ⊘ g) x = f x `mplus` g x

nop :: Rule
nop f = f

many :: Rule → Rule
many r = (r ▷ (many r)) ⊘ nop
```

These operators combine rules at term level, but do not descend into the expression. For that purpose, we define the *once* operator, which applies the rule once somewhere in the expression:

```
once :: Rule → Rule
once r (Comp a b) = (r (Comp a b)) `mplus`
                            (do e ← once r a
                                return (Comp e b)) `mplus`
                                  do e ← once r b
                                     return (Comp a e)
...
```

6.1.3 RL Parser

Although this translation was implemented with the Alloy framework in mind, a parser for RL was also developed. For that purpose, the *Happy* [MG01] parser generator was used. This tool is similar to the *yacc* tool for C, i.e., given a grammar specification, it produces a Haskell program containing a parser for that grammar. The grammar defined for the RL parser is presented in Figure 6.1. We do not consider the input RL expressions to be typed, so it is assumed that all relations are a subset of $U \times U$, where U is a set that contains all possible values.

```
Exp  : Exp && Exp
     | Exp || Exp
     | not Exp
     | Exp ⇒ Exp
     | Var Term Var
     | ( Exp )
     | < forall Var : Exp : Exp >
     | < exists Var : Exp : Exp >
     | Term in Term
     | true
     | false

Var  : varN
     | ( Var , Var )

Term : Term + Term
     | Term & Term
     | Term . Term
     | ˜ Term
     | =
     | bot
     | top
     | pi1
     | pi2
     | id
     | relN
     | < Term , Term >
     | Term \ Term
     | Term / Term
     | ( Term )
```

Figure 6.1: RL parser grammar.

6.1.4 Pretty-printing

The output of the application is a step-by-step RL to CR translation. Although this may be printed directly on the screen, since translations may be long and complex, the result is not very understandable. As such, the tool supports the option of pretty-printing the result in LaTeX. The appearance of the resulting expressions is the same as the one presented in all expressions throughout the thesis.

6.1.5 Core

The base translator takes a RL expression and a rule and applies the later to the former. The expression is passed on a text file, with the format defined by the grammar in Figure 6.1, and the output may either be presented on the screen, or as a LaTeX file. These two functionalities are implemented in the following functions, respectively:

```
transFOL :: String → Rule → IO ()
transFOLTex :: String → Rule → IO ()
```

6.1.6 PW to PF translation

The rules defined for the PW to PF translation include the ones needed for the default translation and for the heuristic simplifications. The translation from section 4.4 is coded under the name `defaultSeq`.

As for the heuristic steps, rules were defined for the Tables 4.3, 4.4 and 4.5 under the names `folRules`, `rlDefs` and `crRules`. These rules may be combined at will, allowing the testing of different approaches. The sequence defined at the end of section 4.6 is implemented on a rule `heuristicSeq`.

6.1.7 PF to PW translation

Translating a PF expression to PW consists mainly in applying the definitions of the RL operators and the basic FOL simplifications. So, considering the reverse definitions of the RL operators is presented in Table 6.1, which we denote by *definitionR*, the rule for a PF to PW translation is defined by:

```
PFPWTrans = insExtVars ▷ many (FOLrules ⊘ definitionR)
```

$a \top b \rightsquigarrow true$	$a\, R \cdot S\, b \rightsquigarrow \langle \exists\, k :: a\, R\, k \wedge k\, S\, b \rangle$
$a \bot b \rightsquigarrow false$	$a\, R \backslash S\, b \rightsquigarrow \langle \forall\, c : c\, R\, a : c\, S\, b \rangle$
$a\, id\, b \rightsquigarrow a = b$	$a\, R / S\, b \rightsquigarrow \langle \forall\, c : a\, R\, c : b\, S\, c \rangle$
$a\, R \cap S\, b \rightsquigarrow a\, R\, b \wedge a\, S\, b$	$(a, b)\, \langle R, S \rangle\, c \rightsquigarrow a\, R\, c \wedge b\, S\, c$
$a\, R \cup S\, b \rightsquigarrow a\, R\, b \vee a\, S\, b$	$(a, b)\, R \times S\, (c, d) \rightsquigarrow a\, R\, c \wedge b\, S\, d$
$a\, \overline{R}\, b \rightsquigarrow \neg a\, R\, b$	$c\, \pi_1\, (a, b) \rightsquigarrow c = a$
$a\, R^{\circ}\, b \rightsquigarrow b\, R\, a$	$c\, \pi_2\, (a, b) \rightsquigarrow c = b$

Table 6.1: Reverse definitions of RL operators (*definitionsR*).

6.2 Alloy to FOL translator

The translation defined in chapter 5 was as implemented in the functional language Haskell. It consists of a parser, also defined using the parser generator *Happy* [MG01]. Before translating the specifications to RL, the model is reduced to a smaller subset, as defined in section 5.1.

The constraints are then translated to RL using the definition of their semantics, defined in 5.4. Then, the RL formulas as passed to the RL to CCR translator, to obtain the PF expressions. The first step is translating the facts to PF. Then, the assertions are translated to PF, using the PF facts along with the properties derived from signature and relation multiplicities as *environment*.

The top-level command translates an Alloy specification from a given file, applies a given rule to the RL expressions resulting from the translation of the specifications, and pretty-prints it in LaTeX. The command is defined as:

```
transAlloyPWPF :: String → Rule → IO ()
```

The final result is presented in a LaTeX file, where the different stages of the translation are presented:

- the original Alloy model;
- the type hierarchy induced by the Alloy model;
- the RL representation of every assertion and fact;
- the facts derived from the signature and relation multiplicity;
- the step-by-step transformation of the RL formulas into CR;
- the final model, consisting on a set of CR inequations.

6.3 Example

We will now present an example of the use of the tool. For simplicity purposes, we will omit the step-by-step PW to PF translation, which is excessively long, and present only the result. The example presented in Figure 6.2 is a very simplified version of the model of a file system used in [FO09], where it is verified both in Alloy and CCR. The file system consists of two relations. The first, `store`, contains the existing files of the system: it relates a set of paths (`Path`), to a set of files (`File`), and is simple: each path points only to a file. The second relation, `open`, defines the files that are open, relating a file handler (`Handle`), to information of the open file (`OpenInfo`), which in this case contains only the path of the opened file. Other than that, each file is related to its type (`type`): a directory (`Dir`) or a regular file (`Reg`), and a path is related to its location by the function dir. We define an assertion, `ri_ok`, that checks the referential integrity: every opened path must exists in the system, and a fact that states that the location of every path must be a directory.

```
sig System {
  store: Path → lone File,
  open: Handle → lone OpenInfo
}

sig Path {
  dir: one Path
}

sig Root extends Path { }

sig OpenInfo{
  path: one Path
}

sig File {
  type: one Type
}

abstract sig Type {}

one sig Reg extends Type {}
one sig Dir extends Type {}

sig Handle {}

fact{
  all s : System, p : Path | some p·(s·store) ⇒ some f : (p·dir)·(s·store) | f·type in Dir
}

assert ri_ok{
  all s : System | all h: Handle, o: h·(s·open) | some f: File | f in (o·path)·(s·store)
}
```

Figure 6.2: A model of a file system in Alloy.

Type hierarchy

$$\begin{aligned} &Type \cong Dir + Reg \\ &Path \cong Path' + Root \\ &univ \cong System + Path' + Root + OpenInfo + File + Dir + Reg + Handle \end{aligned}$$

Relations types

$$\begin{aligned} store &:: System \times (Path' + Rool) \leftarrow File \\ open &:: System \times Handle \leftarrow OpenInfo \\ dir &:: (Path' + Root) \leftarrow (Path' + Root) \\ path &:: OpenInfo \leftarrow (Path' + Root) \\ dir &:: File \leftarrow (Dir + Reg) \end{aligned}$$

Signature multiplicity properties

$$\begin{aligned} \top_{univ \leftarrow univ} \subseteq \top_{univ \leftarrow Dir} \cdot \top_{Dir \leftarrow univ} \wedge \top_{Dir \leftarrow Dir} \subseteq id_{Dir} \\ \top_{univ \leftarrow univ} \subseteq \top_{univ \leftarrow Reg} \cdot \top_{Reg \leftarrow univ} \wedge \top_{Reg \leftarrow Reg} \subseteq id_{Reg} \end{aligned}$$

Relation multiplicity properties

$$\begin{aligned} store^{\circ} \cdot store \subseteq id \\ open^{\circ} \cdot table \subseteq id \\ dir^{\circ} \cdot dir \subseteq id \wedge id \subseteq dir \cdot dir^{\circ} \\ path^{\circ} \cdot path \subseteq id \wedge id \subseteq path \cdot path^{\circ} \\ type^{\circ} \cdot type \subseteq id \wedge id \subseteq type \cdot type^{\circ} \end{aligned}$$

PW Assertions

$$\langle \forall\, s, h, o : (s, h)\ table\ o : \langle \exists\, f, i :: o\ path\ i \wedge (s, i)\ store\ f \rangle \rangle$$

PW Facts

$$\langle \forall\, s, p :: \langle \exists\, k :: (s, p)\ store\ k \rangle \Rightarrow \langle \exists\, f : \langle \exists\, i :: p\ dir\ i \wedge (s, i)\ store\ f \rangle : \langle \forall\, g : f\ type\ g : g\ i_1 \cdot i_1^{\circ}\ g \rangle \rangle \rangle$$

PF Assertions

$$\top \cdot (\pi_1 \cap table \cdot \pi_2) \subseteq \top \cdot (\pi_2 \times id \cap path \bullet \overrightarrow{store}^{2} \cdot \pi_1 \cdot \pi_1 \cdot \pi_1) \cdot \langle id, \top \rangle$$

PF Facts

$$\begin{aligned} \top \cdot store^{\circ} \subseteq (\top \cdot (\pi_2 \times id \cap \cdot dir \bullet \overrightarrow{store}^{2} \cdot \pi_1 \cdot \pi_1 \cdot \pi_1) \cap \\ \top \backslash ((\pi_2 \cdot \pi_1 \times id)^{\circ} \cap \pi_2^{\circ} \cdot \langle type, \overline{i_1 \cdot i_1^{\circ}} \rangle^{\circ}) \backslash \overline{\langle id, \top \rangle}) \cdot \langle id, \top \rangle \end{aligned}$$

Chapter 7

Conclusions and Future Work

Our contribution is an automatic process to translate Alloy models to CCR which is able to embed the Alloy type structure, while still being simple enough for manipulation, on the style of [Oli09]. This represents an improvement over the already existing translation [FPA04], which considered only Alloy formulas, and resulted in extremely complex equations.

RL to CCR translation We defined a complete translation from RL to CCR, whose result is *always* a PF expression. Moreover, we defined a set of heuristic rules which highly simplify the resulting expressions. This transformation is also completely modular and sufficiently generic to be used on any RL expression, and not only the one derived from Alloy. In fact, the option to directly translate a RL formula to CCR is supported by the implementation. The resulting expressions are simple enough so that manual proofs, of the style of [Oli09], can be performed on them. They are also ready to be passed on to the extended static checking tool in development [NOV07], which performs proofs on these kind of formulas.

Alloy to RL translation We presented an Alloy to RL translation that considers Alloy models as whole: properties and a type hierarchy are derived from the signature and relation declarations, which, allied to the facts from the Alloy model, provide an environment for the translation of its assertions. This represents an improvement from the already existing translation [FPA04], which only considered Alloy formulas. However, we still do not consider the whole Alloy core. Particularly, the closure operators and the comprehension, are left out because there is no way to simply represent them in CCR. Also, by trying to represent Alloy's type system in CCR, a complicated, albeit correct, translation resulted. Fortunately, in most cases the result ends up being not that complex, due to the PF simplifications performed afterwards.

N-ary operators We defined a new n-ary rotation operator, $\overrightarrow{R}$, adapted the n-ary composition $\bullet$ defined in [FPA04] to our framework, and defined a set of properties over those operators. By enhancing the RL to CCR translation with these operators, we were able to completely remove all quantifiers from the expression and still obtain formulas that are simple enough to understand and manipulate. However, although we have derived a set of useful properties about these operators a more exhaustive search for properties must still be performed.

With this process, we provide a way to perform *unbounded* verification of Alloy models, whenever the model checking performed by the Alloy Analyzer, within a limited scope, is not enough.

Future work We did not present any proof made on the PF expressions that result from the translations. Studying how simple manipulating and verifying those expressions is is an essential step to validate the results of this project. This will directly depend on the richness of properties possessed by the new n-ary operators. As such, a more deep analysis of the operators is needed,

to assess on their utility. If the presence of n-ary operators highly complicates the manipulation of inequations, there is the need to consider if CCR, or at least the way we represented n-ary RL in it, is the best framework to deal with n-ary relations.

Improving the translation so that it is able to process closure operators and comprehension would be an important step. However, it would probably highly increase the complexity of the final expressions, so there is a need to study the consequences of that enhancement.

We also consider if the approach we took on the types, implementing them in a categorical, hence typed framework, was the best course of action. The other possibility was leaving CR untyped (using FA, or CCR with only one type containing all values), and define the types as subsets of the universe type. Subsets in CR are represented by correflexives, so our type system would consist of a set of inequations involving correflexives. This way, the PF expressions could be checked on existing theorem provers for untyped RL, such as the approach took on [HS08], where many CR theorems are automatically verified by theorem provers.

Another possibility, if the PF calculus proves not to be feasible, is to embed our RL expressions, representing the semantic of the Alloy, in a higher-order theorem prover, like Isabelle [Pau94]. There already exist some projects which embed modeling languages in higher-order theorem provers with successful results [BRW03, Ver07].

There is a project in course that consists on the development of a formal methods tool-chain, in which the different tools, one of which Alloy, are connected by the point-free relational calculus [FO09]. However, this translation is presently made manually, which, not only is very time consuming, but also error prone. A more short-term contribution of this translation is enhancing the tool-chain with an automatic Alloy to CCR conversion.

Bibliography

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[Bac78] John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[Bac04] Roland C. Backhouse. Mathematics of program construction. Draft of book in preparation, 2004.

[BdM96] Richard Bird and Oege de Moor. *Algebra of Programming*, volume 100 of *International Series in Computer Science*. Prentice-Hall, Inc., 1996.

[BFHL96] Gabriel Baum, Marcelo F. Frias, Armando M. Haeberer, and Pablo E. Martínez López. From specifications to programs: A fork-algebraic approach to bridge the gap. In *MFCS '96: Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science*, pages 180–191. Springer-Verlag, 1996.

[BJ78] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer-Verlag, 1978.

[BRW03] Achim D. Brucker, Frank Rittinger, and Burkhart Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *The Journal of Universal Computer Science*, 9(2):152–172, 2003.

[CPP06] Alcino Cunha, Jorge Sousa Pinto, and José Proença. A framework for point-free program transformation. In *Revised Papers of the 17th International Workshop on Implementation and Application of Functional Languages (IFL'05)*, volume 4015 of *LNCS*, pages 1–18. Springer-Verlag, 2006.

[EJT04] Jonathan Edwards, Daniel Jackson, and Emina Torlak. A type system for object models. *SIGSOFT Softw. Eng. Notes*, 29(6):189–199, 2004.

[EM45] Samuel Eilenberg and Saunders MacLane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, 58(2):231–294, 1945.

[FA94] Marcelo F. Frias and N. G. Aguayo. Natural specifications vs. abstract specifications. a relational approach. In *Proceedings of SOFSEM'94*, pages 17–22, 1994.

[FAN93] Marcelo F. Frias, N. G. Aguayo, and B. Novak. Development of graph algorithms with fork algebras. In *Proceedings of the XIX Latinamerican Conference on Informatics*, pages 529–554, 1993.

[FHV97] Marcelo F. Frias, Armando M. Haeberer, and Pablo A. S. Veloso. A finite axiomatization for fork algebras. *Logic Journal of the IGPL*, 5(3), 1997.

[FO09] Miguel A. Ferreira and José N. Oliveira. An integrated formal methods tool-chain and its application to verifying a file system model. In *Formal Methods: Foundations and Applications, 12th Brazilian Symposium on Formal Methods, SBMF 2009*, volume 5902 of *LNCS*, pages 153–169. Springer-Verlag, 2009.

[FPA04] Marcelo F. Frias, Carlos López Pombo, and Nazareno Aguirre. An equational calculus for Alloy. In *6th International Conference on Formal Engineering Methods*, volume 3308 of *LNCS*, pages 162–175, 2004.

[FPM07] Marcelo F. Frias, Carlos López Pombo, and Mariano M. Moscato. Alloy Analyzer + PVS in the analysis and verification of Alloy specifications. In *TACAS*, volume 4424, pages 587–601. Springer-Verlag, 2007.

[Fri02] Marcelo F. Frias. *Fork Algebras in Algebra, Logic and Computer Science*, volume 2 of *Advances in Logic*. World Scientific Publishing Co., Inc., 2002.

[Fv90] Peter J. Freyd and Andre Ščedrov. *Categories, allegories*, volume 39 of *North Holland Mathematical Library*. North Holland, 1990.

[GM93] Michael J. C. Gordon and Tom Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Gyu95] Viktor Gyuris. A short proof for representability of fork algebras. *Logic Journal of the IGPL*, 3(5), 1995.

[HBS93] Armando M. Haeberer, Gabriel Baum, and Gunther Schmidt. On the smooth calculation of relational recursive expressions out of first–order non–constructive specifications involving quantifiers. In *Proceedings of the International Conference on Formal Methods in Programming and Their Applications*, pages 218–298. Springer–Verlag, 1993.

[HS08] Peter Höfner and Georg Struth. On automating the calculus of relations. In *Proceedings of Automated Reasoning, 4th International Joint Conference, IJCAR 2008*, volume 5195 of *LNCS*, pages 50–66. Springer-Verlag, 2008.

[HV91] Armando M. Haeberer and Pablo A. S. Veloso. Partial relations for program derivation: Adequacy, inevitability and expressiveness, in constructing programs from specifications. In *Proceedings of the IFIP TC2 Working Conference on Constructing Programs from Specifications*, pages 319–371, 1991.

[Jac02] Daniel Jackson. Alloy: a lightweight object modeling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[JW96] Daniel Jackson and Jeannette Wing. Lightweight formal methods. *IEEE Computer*, 29(4):21–22, 1996.

[LÏ5] Leopold Löwenheim. Uber möglichkeiten im relativkalkul. *Mathematische Annalen*, 76:447–470, 1915.

[LV02] Ralf Lämmel and Joost Visser. Typed combinators for generic traversal. In *Proceedings of the Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, page 137–154. Springer-Verlag, 2002.

[Mad06] Roger Duncan Maddux. *Relation Algebras*. Studies in Logic and the Foundations of Mathematics. Elsevier, 2006.

[MG01] Simon Marlow and Andy Gill. *Happy User Guide*, 2001.

[Mor60] Augustus De Morgan. On the syllogism, no. IV. and on the logic of relations. *Transactions of the Cambridge Philosophical Society*, 10:331 – 358, 1860.

[MSS92] Szabolcs Mikulàs, Ildikó Sain, and András Simon. Complexity of the equational theory of relational algebras with projection elements. *Bulletin of the Section of Logic*, 21(3):103 – 111, 1992.

[NOV07] Claudia M. Necco, José N. Oliveira, and Joost Visser. Extended static checking by rewriting of point-free relational expressions. 2007.

[OCV06] José N. Oliveira, Alcino Cunha, and Joost Visser. Type-safe two-level data transformation. In *14th International Symposium on Formal Methods*, volume 4085 of *LNCS*, pages 284–299. Springer-Verlag, 2006.

[Oli08] José N. Oliveira. Transforming data by calculation. In *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *LNCS*, pages 134–195. Springer-Verlag, 2008.

[Oli09] José N. Oliveira. Extended static checking by calculation using the point-free transform. In *LerNet ALFA Summer School 2008*, volume 5520 of *LNCS*, pages 195–251. Springer-Verlag, 2009.

[OR06] José N. Oliveira and César Rodrigues. Point-free factorization of operation refinement. In *14th International Symposium on Formal Methods*, volume 4085 of *LNCS*, pages 236–251. Springer-Verlag, 2006.

[Ore44] Oystein Ore. Galois connexions. *Transactions of the American Mathematical Society*, 55:493–513, 1944.

[Pau94] Lawrence C. Paulson. *Isabelle - A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[Pei33] Charles S. Peirce. Note B. the logic of relatives. In *Studies in logic by members of the Johns Hopkins University*, pages 187 – 203. Little, Brown and Company, Boston, 1833.

[Sch95] Ernst Schröder. *Vorlesungen uber die Algebra der Logik (exakte Logik)*, volume 3, part 1: Algebra und Logik der Relative. Leipzig, 1895.

[Spi89] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., 1989.

[Tar41] Alfred Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6(3):73 – 89, 1941.

[TG87] Alfred Tarski and Steven Givant. *A Formalization of Set Theory without Variables*, volume 41 of *Colloquium Publications*. American Mathematical Society, 1987.

[Ver07] Sander D. Vermolen. Automatically discharging VDM proof obligations using HOL. Master's thesis, Radboud University Nijmegen, 2007.

[VH91] Pablo A. S. Veloso and Armando M. Haeberer. A finitary relational algebra for classica first-order logic. *Bulletin of the Section of Logic*, 20(2):52 – 62, 1991.

[VH93] Pablo A. S. Veloso and Armando M. Haeberer. On fork algebras and program derivation. Technical Report MCC 32/93, Departamento de Informática, PUC-Rio, 1993.

[Wad92] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14. ACM Press, 1992.