Translating Alloy to SMT-LIB

by

Forrest C. Cinelli, Kyle D. McCormick

A Major Qualifying Project

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

in Computer Science and Mathematical Sciences

by

March 2018

APPROVED:

Daniel J. Dougherty

Abstract

Alloy is a tool for writing specifications and constructing instances of these specifications, based on relational logic. Satisfiability Modulo Theories (SMT) solvers embody another popular approach to specification and instance-generation; most solvers implement a language based on the SMT-LIB standard. The Alloy language and the core of SMT-LIB are each formally equivalent, over finite structures, to first-order mathematical logic; however, they support quite different modeling idioms. To help bridge this gap we have initiated a project to construct a translation from Alloy to SMT-LIB.

This document is a report on a the first stage of this work: a formal definition of the translation, a proof-of-concept implementation, and some preliminary performance measurements.

Acknowledgements

We would like to extend our sincerest gratitude to Professor Dougherty, whose mentorship, patience, and expertise made this project possible. Professor Dougherty consistently went out of his way to make sure that we were challenged and engaged by the project work, and took personal interest in helping us grow as researchers. When a part of the project proved more difficult than expected, Professor Dougherty was understanding and helpful, but still firm as a teacher, giving us the time and resources we needed to solve a problem rather than simply giving us the answer.

Furthermore, we would like to thank all the family, friends, and teachers who have supported and encouraged us in our academic endeavors over the years. Kyle would also like to specifically thank his high school teacher Mr. John Lecce, who kindled his interest in computer science and pure mathematics.

Contents

Chapter 1

Introduction and Motivation

Alloy comprises a language for writing logical specifications and a tool for analyzing those specifications. Constraints are expressed in the Alloy language using relational algebra. The Alloy Analyzer is the tool that analyzes specifications written in the Alloy language. It works by translating an Alloy language specification into propositional logic, which it then passes to a SAT solver.

Satisfiability Modulo Theories (abbreviated SMT) problems are Boolean satisfiability problems defined with respect to a fixed background theory. The SMT-LIB language is the industry standard for capturing SMT problems, and many tools exist to solve SMT problems specified in the SMT-LIB language.

Alloy and SMT-LIB each have their own tool ecosystem and community. By working towards interoperability of the two languages, we hope to bridge this gap, encouraging collaboration between their communities. For example, by making SMT solvers a viable backend for Alloy, Alloy users may then be able to take advantage of tools that were only previously available to SMT-LIB users.

Another difference between Alloy and SMT is that Alloy does its analysis relative to a user-defined finite scope, which means that it cannot prove that no counterexample to a statement exists, or that a specification definitely has no instances at all. This method "limits the analysis to models with explicit and concrete cardinality bounds on the relations involved; hence it is appropriate only for proving the consistency of a model or disproving that a given property, encoded as a formula, always holds for a model" [MRTB17]. SMT solvers do not have these limitations, so it is natural to want to analyze Alloy relational algebra using an SMT solver rather than a SAT solver.

SMT solvers also have access to more background theories, such as background theories for real number arithmetic or collections like arrays or sets, so with further work the capabilities of the Alloy language could be expanded even more.

Finally, SMT solvers like Z3 are highly optimized, achieving fast runtimes in a wide variety of situations. It would interesting to explore whether using SMT solvers as a backend to the Alloy language could lead to significant performance improvements over the Alloy Analyzer. A translation such as the one developed here can help explore this question.

1.1 Model Finding

For critical applications like billing systems, firmware, and security, it is useful to be able to prove facts about the specification of an application and its behavior. While reasoning generally about computer programs themselves is undecidable, human users can write a formal theory that is an abstraction of a system, and attempt to prove or disprove assertions about this abstraction. Given a first-order theory that expresses known properties of some system, and a statement that a user hopes holds about the system, one can use an automated deduction system to determine whether the given statement holds in the presence of the given theory, i.e. whether the theory *entails* the statement.

An alternative approach is to explore the *models* of the given theory. *Satisfiability* is capable of everything automated deduction is, since checking whether a theory entails a statement is equivalent to checking that the conjunction of the theory and the negation of the statement has no models. Formally, satisfiability is determining whether there exist models of a theory. A formula is *satisfiable* if a model does exist, and *unsatisfiable* otherwise [SLM+92].

Model finding, however, can be even more useful. Unlike satisfiability, which is determining whether models of a theory *exist*, model finding is actually searching for and reporting specific models. Finding and displaying specific models has several benefits. Most obviously, if a user checks a property and that property fails unexpectedly, it can be helpful to see example situations in which the property failed. But even beyond that, model finding can be used to organically explore the possibilities left open by a theory. For example, if a user has a theory that represents a building access policy, and wants to know "who can enter the maintenance tunnels after 11pm?" they could search for models of their access theory and the statement "a person enters the maintenance tunnels after 11pm." Such an exploration will often uncover situations the user expected, but may also yield models of situations that the user wasn't aware were possible, or even that the user intended to be impossible. This kind of exploration of a formal specification of the behavior of a system is the core offering of model finding.

An important and direct application of satisfiability is logical consequence: to show that a statement a entails another statement b, it suffices to show that the formula $a \wedge \neg b$ is unsatisfiable, that is, in the presence of a, $\neg b$ is never true, which means that a entails b. This problem, like satisfiability itself, is undecidable [Tra50]. However, logical consequence *is* semi-decidable, so satisfiability is co-semi-decidable.

In the next sections, we describe the parts of the field of model finding relevant to our work.

1.2 SMT and SMT-LIB

The Boolean satisfiability problem, often referred to as the SAT problem, is determining whether there is a model that satisfies a given Boolean formula, expressed in propositional logic. For example, the SAT problem from $a \vee b$ is satisfiable, and one model witnessing this is the one that sets a to true and b to false. Satisfiability Modulo Theories (a.k.a. SAT Modulo Theories, abbreviated SMT) problems are SAT problems with respect to a fixed background theory. For example, in the problem $x < y$ for x and y integers, it's likely that one would rather have the meaning of $<$ be fixed to always refer to the function that orders integers, rather than letting it range over all possible meanings a SAT solver might consider.

When an SMT problem has the background theory which makes no restrictions on any objects, sometimes called the 'theory of equality' or the 'theory of equality with uninterpreted functions,' the problem is simply SAT for first-order logic. In general, this problem is undecidable. [BSST09]

SMT-LIB is a research initiative aimed at facilitating research into SMT. Their most interesting product for our purposes here is the SMT-LIB 2.6 language which the SMT-LIB organization writes the specification for. The SMT-LIB language maps directly to first-order logic, unlike the relational semantics of Alloy. As in Alloy, the building blocks of SMT-LIB theories are terms, but a term can be of any type, including Boolean type [C+]. It will sometimes be useful to refer to a boolean-valued SMT-LIB term as a *formula*, only for brevity. For a complete and rigorous description of Version 2.6 of the SMT-LIB language, see [BFT17].

1.3 Alloy

Alloy is a language and a tool. The Alloy language expresses a logic based on relations and relational algebra, and it can model an inheritance hierarchy among its different *signatures*, which are sets akin to SMT-LIB sorts. A specification written in the Alloy language is often called a *model* in the Alloy community, but we will refer to one as a "specification," rather than as a model, to avoid confusion with the definition of "model" used in the field of satisfiability and model finding, as described above. Furthermore, in the Alloy community, the mathematical thing known as a "model" is called an *instance*.

The Alloy Analyzer is a tool for analyzing Alloy specifications. It translates an Alloy specification into a SAT problem, which is then solved by the Kodkod SAT solver [Jac12a]. The Alloy Analyzer is a model finder: it searches for instances in which the theory and a given statement are true. It can easily be used to find counterexamples to a statement as well by simply negating the statement and then proceeding in the same way.

In order to ensure decidability, the Alloy Analyzer performs model finding over a restricted *scope*, that is, it limits its search to models with fewer unique objects than a given number. Alloy applies this restriction on a per-signature basis: each individual signature has a maximum number of elements. The user can specify this scope for each signature, and the Analyzer will use a default value for the number of members of any signature whose size wasn't restricted by the user [Jac12b]. This means that the analyzer may reject as possibly inconsistent a specification which can have models, and may fail to find a counterexample for a specification that has one.

The fundamental building blocks of an Alloy specification are terms. A term is either a formula or an expression. A *formula* is a boolean-valued term, and an *expression* is a non-boolean-valued term.

An Alloy specification is composed of a list of syntactic constructs, each of which is one of the following:

- **Sigs**, which denote sets of elements.
- **Relation declarations**, which are, in the concrete syntax, bundled with sig declarations.
- **Facts**, which are formulas. The Alloy Analyzer will only report instances which satisfy the conjunction of all the facts in the Alloy specification it is analyzing.
- **Assertions**, which are formulas that can be referenced by commands.
- **Functions**, which are a list of variable declarations which are the function's parameters, and a body which is any expression.
- **Predicates**, which are functions which have a formula as their body rather than an expression.
- **Commands**. A command is either a **`run`** or a **`check`**. A command takes a formula, a reference to a function, or a reference to an assertion or predicate.

  A **`run`** command searches for an instance where the given formula or assertion or predicate is true, while a **`check`** command searches for an instance which is a counterexample to the given formula or assertion or predicate.

For a complete discussion of the grammar and syntax of the Alloy language, see [Jac12b], appendix B of [Jac12a].

Chapter 2

Design Decisions

2.1 Design: Mathematical Concerns

Two languages were introduced as intermediaries in the translation from Alloy to SMT-LIB. The first of these is called **Core Alloy**, and is a simplified version of Alloy with the same semantic power. Its specifics are primarily software engineering concerns, and are therefore discussed in later sections. The second of these is called the **Intermediate Language**.

2.1.1 Relations as Boolean Valued Functions

Relations abound in Alloy, but they are not directly modeled by any SMT-LIB structure. However, relations are logically equivalent to Boolean valued functions, so our translation uses them to represent Alloy relations.

2.1.2 Core Alloy

The Alloy language has many groups of constructs that are similar to one another. For example, the command **`check`** searches for a model that is a counterexample to a given formula, and the command **`run`** searches for a model that satisfies the specification and the given formula. But the command **`check`**x is logically equivalent to **`run`**$\neg x$, so rather than individually translate both of these concepts, it is simpler to first express every instance of one in terms of the other. Additionally, syntactic sugar in Alloy can be removed before translation. For example, the inheritance hierarchy of Alloy sigs can be expressed in a list of facts rather than a group of syntactic constructs around the sig declarations.

The purpose of defining a core was to simplify the translator, so we chose whether or not to include different constructs in the core or not based on how parallel to SMT-LIB (the target language) a feature of the Alloy language is. We could have simplified Core Alloy much more: in Appendix C of Software Abstractions [Jac12a], Jackson defines the Alloy kernel, which is the smallest possible subset of Alloy which can capture the functionality of the whole language. Its grammar is only a few lines long. We chose not to reduce Alloy this much because it was ultimately unnecessary: we would sacrifice readability of the generated code without significantly simplifying our pipeline (simply moving implementation complexity from the translator to the reduction to Core Alloy). Striking a balance in how much to simplify Alloy during the translation to core was an interesting challenge, and one we think we met well. The translator to core removes a lot of complexity, but Core Alloy is still readable and expressive, and ultimately Core Alloy is much more similar to SMT-Lib than Alloy is. In section 3.2 we describe in detail which structures are included in the Alloy core and which are not.

2.1.3 The Intermediate Language

We decided early on that it would be better to divide the translation from Core Alloy to SMT-LIB into two steps: first, we would translate Alloy into a higher order logic, and then reduce the higher-order logic representation to SMT-LIB. This is helpful because Alloy is (on the surface) second-order; that is, variables in Alloy may be *relation-valued*, while variables in SMT-LIB, are strictly scalars. A single Alloy variable may be thought of as a table, whose number of rows equals the cardinality of the relation, and whose number of columns equals the arity of the relation. While these constructions are more complex, the Alloy analyzer rejects constructions that are truly not first-order, and so does our translation. Stepping into higher order logic during the translation affords us the mechanical flexibility needed to represent the complex objects that are Alloy variables as more elementary expressions that are easily expressed in SMT-LIB.

We chose to make the Intermediate Language as close to SMT-LIB as possible. In fact, to define the Haskell data structures to represent the IL, we copied the SMT-LIB data structures from the SMT2Lib library which we used later in the pipeline to represent SMT-LIB, and modified them to be higher order. Ultimately, the Intermediate Language simply became SMT-LIB with abstraction terms and tuple terms, so reduction from the Intermediate Language to SMT-LIB is simply beta reducing every redex (a *redex* is an application of an abstraction to some arguments) and breaking apart tuples. In this scheme it is also easy to determine whether our translator's output is first-order: to do so, simply check that every abstraction term is contained within a redex.

Specifying the Intermediate Language further simplified the transformation that the translator itself needed to do, which also simplified and streamlined the code of the translator, and ultimately allowed the entire pipeline to be more straightforward and simple.

2.1.4 The Universal Sort, and Sigs as Unary Predicates

The Alloy language has *sigs*, which are sets containing elements. Membership of scalars in sigs can be tested easily, and functions and relations can have domain or range restricted to certain sigs in their definition, for example `**fun** f [a:S]: T`, whose parameter `a` must be from sig `S` and whose range must be contained in `T`[1]. The natural analogue of Alloy sigs in SMT-LIB is sorts, which are used in very similar ways. However, SMT-LIB sorts must be disjoint. While "top-level" Alloy sigs must all be disjoint, Alloy sigs can have complex, arbitrary relationships between a given sig and all of its "children," other sigs defined with extra syntax reminiscent of inheritance patterns in object-oriented programming languages like Java. Formally, *top-level* sig is simply a sig that does not **`extends`** another sig and is not **`in`** another sig, and a *child* sig is one that does **`extends`** another sig or is **`in`** another sig. For example, the Alloy code

```
sig S{}
sig A, B extends S{}
```

Has three sigs `A`, `B`, and `S`, and also states that `A` and `B` are (1) subsets of `S`, and (2) disjoint from each other.

Because SMT-LIB sorts are always disjoint from each other, they cannot be used to model inheritance relationships between Alloy sigs. SMT-LIB does, however, allow *unary predicates*, which are functions that take one argument and return a Boolean. Mathematically, a sort is simply a set, and a unary predicate is a function that induces a set, i.e. the set of all arguments it is true for. So broadly, there are two options: (1) model top-level Alloy sigs as sorts, and model sub-sigs with unary predicates, or (2) have one universal sort (which every scalar is a member of) and model *all* sigs from the input Alloy specification as unary predicates. We chose option (2) because the uniformity of that approach made it simpler to implement, both mathematically and programmatically.

2.2 Design: Software Engineering Concerns

Pipeline Architecture

[1] In reality, the exact meaning of these type notations is a little more complex. See [Jac12a], especially [Jac12b], for details

As we thought about the structure the code would have to take to solve the problem, and the behavior we wanted it to have, organizing the translator into a pipeline became the natural choice. The first step in translating any programming language into another is lexing and parsing the text representing the input language. Additionally, as we thought about the how the translation itself would work from a mathematical perspective, it became clear that we wanted to express the meaning of Alloy specifications in higher order logic, which we would later reduce to first order logic (which was possible since relational algebra, which Alloy is based on, is also first order), rather than move directly to a first order representation. Finally, we also saw the value in specifying a simplified core of Alloy, expressing complicated syntactic constructs in terms of other, simpler parts of Alloy, and then translating this simplified specification. The result of this early thinking was that we all but fell into the idea of a multi-stage pipeline: the translation was clearly going to be a series of transformations, so a pipeline architecture was the most obvious choice to proceed with.

One disadvantage of a pipeline architecture is that it introduced some redundancy into the code. Every stage of the pipeline needs to traverse the entire tree representing a specification, but may not actually need to modify most terms. The normalizer is the most egregious example of this: it has to explore the entire tree, but except for abstraction terms it leaves every other term untouched! However, we think that for this project, the benefits of the pipeline architecture outweigh the drawbacks. The redundant code is not very complex, so it is not a major source of errors, nor does it significantly impact the execution time of the complete program. The separation of concerns, code simplification, and debugging advantages provided by the pipeline architecture more than justify this drawback.

The pipeline we implemented has six stages:

1. **Lexing**, in which input Alloy source code is split into tokens. The Haskell code for

this was generated by the Alex lexer generator.

2. **Parsing**, in which the tokens are parsed into an abstract syntax tree. The Haskell code for this was generated by the Happy parser generator. No type- or arity-checking is done in this stage, since Happy is not equipped to do them.

3. **Translation to Core Alloy**. Core Alloy is a smaller yet semantically equivalent subset of Alloy. As it is generally easier to write and maintain two distinct, smaller algorithms than it is to maintain one large one, we chose to section off Core Alloy from the language as a whole. By reducing to a core source language, we minimize the size and complexity of the code necessary to translate to the target language. We believe this decision made the next pipeline stage, simpler, less redundant, and easier to code robustly than it would have been otherwise.

   In addition to reducing to the Core, this pipeline stage also differentiates between Alloy formulas (boolean-valued terms) and expressions (non-boolean-valued terms) and checks that arguments to operators have valid arities. This metadata is built in to the Core Alloy AST data structure.

   Core Alloy and the translation to it are described in depth later in this chapter.

4. **Translation to the Intermediate Language**. The Intermediate Language (IL) is a higher-order, simply-typed, lambda-calculus that is an abstraction over both Alloy and SMT, enabling us to bridge the gap between the two languages. It treats Alloy expressions as abstractions that return Booleans. Furthermore, the only types in the language are booleans, univ (the set of non-boolean scalars), tuples, and functions. The details of the IL are described in greater detail later in this section, and some mathematical properties of the IL are discussed in Section 4.

5. **Normalization**. All *redexes* (applications of abstractions to arguments) are *beta-*

reduced (the act of substituting in the arguments to an anonymous function for its parameters). This is done using a tree traversal that should, for all valid input Alloy specifications, leave no abstractions in the normalized IL. Normalization is described in more detail later in this section.

6. **Translation to SMT**. Because the IL is already very similar to SMT, this is a simple transformation that involves primarily lowering tuples into separate scalar values. This stage is described in more detail later in this section.

Finally, after each stage of processing, the resulting SMT abstract syntax tree is formatted and printed to standard output.

Chapter 3

Preliminaries

In this chapter we will discuss four languages that roughly correspond to the stages of the translation: Supported Alloy, Core Alloy, the Intermediate Language (the IL), and SMT-LIB. Supported Alloy is a subset of the Alloy 4 language; we will show its grammar and give notation to represent the components of a Supported Alloy specification. Core Alloy is a transformed version of Supported Alloy that is desugared, requires fewer syntactic constructs, and has fewer operations; it is designed to simplify Alloy in order to decrease the complexity of the translation to IL. The IL is a simply-typed lambda calculus which allows the translation to generate fragments which are higher-order. However, the translation of a formula is still guaranteed to be first-order, so any IL the translation generates can be lowered into normal form. Once this is done, it can be translated into SMT-LIB, the destination language.

Of these four languages, we will give precise definitions for Supported Alloy, Core Alloy, and the IL. Once normalized, the IL is very similar to a subset of SMT-LIB, making the transformation very straightforward. Hence, we omit a full mathematical definition of SMT-LIB and simply refer the reader to [BFT17], the SMT-LIB language standard.

3.1 Supported Alloy

The translation described in this paper handles a subset of the Alloy language, which we refer to as *Supported Alloy*. In this section we outline the features of Alloy our translator does support and does not support. We will attempt to justify more interesting choices. For a more detailed description of any of the parts of the Alloy language listed below, see [Jac12b], the Alloy language reference. At the time of writing this reference is available for free online.

3.1.1 Supported Subset of Alloy

We support the following operators and syntactic constructs of the Alloy Language:

- Defining functions and predicates.
- Functions and predicates with multiplicity and/or disjointness constraints in parameters. For example, the function declaration ⟨`f [disj a, b: S]`⟩ is supported.
- Defining sigs, including:
  - Sigs that are **`abstract`**, or **`extends`** or are **`in`** other sigs.
  - Sigs with multiplicity constraints such as **`one`** or **`some`**.
  - Defining relations with *simple* multiplicity constraints (that is, multiplicity constraints that are **not** embedded within Cartesian product expressions).
  - Defining relations as disjoint, e.g. ⟨`sig S{disj u, v : S}`⟩.
- Defining facts.
- Defining assertions.
- Running blocks, running predicates, checking blocks, and checking assertions.

- User-defined scopes on runs and checks.
- The special relations **`iden`**, **`none`**, and **`univ`**.
- Formula blocks, which we translate as the conjunction the translation of every formula in the block.
- Quantification, i.e. expressions of the form $\langle$`{m x: e | f}`$\rangle$, where *m* is a multiplicity keyword (**`no`**, **`lone`**, **`some`**, **`one`**, or **`all`**) *x* is a variable of type *e*, and *f* is a formula. The translator understands that these expressions are formulas checking that the members of the set $\langle$`{x: e | f}`$\rangle$ satisfy *m*.
- Set constructors, including declarations with the**`disj`** keyword. These are of the form $\langle$`{x : e | f}`$\rangle$, where *x* is a variable of type *e*, and *f* is a formula. The translator understands the meaning of these expressions as the set containing every element of *e* for which *f* is true, which is different from quantification formulas.
- Multiplicity formulas, e.g. $\langle$**`some`** `x`$\rangle$
- Box join and dot join.
- **`let`** expressions and formulas.
- **`implies-else`** expressions and formulas.
- Product expressions *without* multiplicities, e.g. $\langle$`S ->` `T`$\rangle$, but not $\langle$`S` **`one ->`** `T`$\rangle$ and not $\langle$ `S` **`-> some`** `T`$\rangle$.
- Cardinality formulas of the form $\langle$**#** *S* **<=** *n*$\rangle$ or $\langle$**#** *S* **=** *n*$\rangle$, where *S* is a sig and *n* is a natural number.

- The operations union (denoted `+`), difference (denoted `−`), intersection (denoted `&`), domain restriction (denoted `<:`), range restriction (denoted `:>`), and relational override (denoted `++`).
- The operators **`and`**, **`or`**, **`implies`**, and **`iff`**.
- The subset operator (denoted **`in`**) and equality operator (denoted `=`).
- Function and predicate application though box join, dot join, or some combination of the two.

3.1.2 Unsupported Features

We chose to not translate certain features of Alloy for one of three reasons. Firstly, we saw some features of Alloy as outside the core objective of this Major Qualifying Project (MQP); we ignored issues like modules, imports, and enumerations, which are more "quality-of-life" features than interesting mathematical concepts. Secondly, there were some other features that, although mathematically interesting, we deemed beyond the scope of this MQP given the time project's time constraints. Finally, there are a few features we were capable of doing and planned to support, but simply ran out of time to do. We hope to revisit those features later, and below we describe what we would do to implement each of them in our translator.

Outside the Objective of This MQP

- Modules, importing and exporting, and the **`private`** keyword. The primary goal of this MQP was to investigate the mathematical relationship between Alloy and SMT. We mention in Future Work that if this translator were to one day become a robust, professionally-used tool, support for modules should be added.

- Qualified names (names of the form "`namespace/x`"), for similar reasons the previous item.

- Enumerations, because they do not add new mathematical meaning to the language. Enumerations have a clear analogue in SMT-LIB, so implementing them in the translator would be straightforward.

Out of Scope

- Quantification over sets (e.g. ⟨**`all`** `a:`**`set`** `A | ...` ⟩), which is not necessarily first-order. Alloy only allows this in special situations which its Analyzer can handle.

- Arithmetic, which is simply not a primary focus of Alloy.

- Transitive closure, which cannot be fully expressed in first order logic[EGTH15].

- Sequences. We simply decided that the sequence data type was outside the main scope of comparing relational algebra to first order logic, so what we estimated would be a large amount of work would be better spent focusing on the core of the Alloy language. This is another feature which would be important for the translator to be a fully fledged tool, rather than just an experiment as it is now.

- Arbitrary cardinality formulas. The translator currently supports cardinality formulas of the form ⟨**#** *a* **=** *b* ⟩ or ⟨**#** *a* **<=** *n* ⟩, for a sig *a* and an integer *n* It would be straightforward to extend this to include operators **<**, **>** and **>=**, and generalize *a* to all expressions. However, cardinality formulas of the form ⟨**#** *a* *op* **#** *b* ⟩, for some comparator *op* and expressions *a* and *b*, would be more challenging. We concluded that these formulas are not related to the core functionality of Alloy and so could be omitted. The problem they pose is interesting, however, and could be the subject of some enlightening future work.

Abandoned Because of Time Constraints

- Product expressions with multiplicities. These are expressions of the form $\langle a \;\; m$ **->** $n \;\; b \, \rangle$, where a and b are expressions and m and n are multiplicities. Note that a and b can themselves be of the form $\langle a \;\; m$ **->** $n \;\; b \, \rangle$, so a translation would have to handle the case where these expressions are nested. Right now these expressions are not in Core Alloy, and we believe that the best way to handle them would be somehow extracting the multiplicity constraints during the translation to Core Alloy.

- Defining predicates or functions using *receiver notation*. Handling this would simply be a matter of expressing the rules for Alloy receiver notation in the Haskell code so that the translator can correctly infer when items should be implicitly joined with the parent sig.

- The keyword **`this`**, which is closely related to receiver notation.

- Signature facts. The transformation from sig facts to top-level facts would be trivial if the translator understood receiver notation.

- Gracefully handling multiple **`run`** or **`check`** commands. Solving this problem would involve deciding what the behavior of the translator *should* be in addition to actually implementing it. Supporting this feature would be vital in bringing the translator from a research experiment to a robust tool.

One final restriction of Supported Alloy is that it must have exactly one command (that is, exactly one paragraph in the form $\langle$**`run`** ... $\rangle$ or $\langle$**`check`** ... $\rangle$).

The grammar of Supported Alloy is defined below. Note that some of the rule names are underlined; this is explained in 3.1.2.

Definition 3.1.1 (Grammar of Supported Alloy)**.**

$\underline{spec}\ ::=\ para\star$

$para\ ::=\ sigDecl\ \big|\ predDef\ \big|\ funDef\ \big|\ assert\ \big|\ fact\ \big|\ cmd$

$\underline{sigDecl}\ ::=\ \big(\ abstract?\ sigMult?\ \big|\ sigMult?\ abstract?\ \big)\ \textbf{\texttt{sig}}\ \ names\ extClause?\ \{\ decls\ \}$

$abstract?\ ::=\ \big[\ \textbf{\texttt{abstract}}\ \big]$

$sigMult?\ ::=\ \big[\ \textbf{\texttt{lone}}\ \big|\ \textbf{\texttt{some}}\ \big|\ \textbf{\texttt{one}}\ \big]$

$\underline{extClause?}\ ::=\ \big[\ \textbf{\texttt{extends}}\ \ name\ \big|\ \textbf{\texttt{in}}\ \ name\ \big(\ \textbf{\texttt{+}}\ \ name\ \big)*\ \big]$

$\underline{predDef}\ ::=\ \textbf{\texttt{pred}}\ \ name\,[decls]\ block\ \big|\ \textbf{\texttt{pred}}\ \ name\,(decls)\ block$

$\underline{funDef}\ ::=\ \textbf{\texttt{fun}}\ \ name\,[decls]\ :\ expr\ \{\ expr\ \}\ \big|\ \textbf{\texttt{fun}}\ \ name\,(decls)\ :\ expr\ \{\ expr\ \}$

$\underline{assert}\ ::=\ \ \textbf{\texttt{assert}}\ \ name\ block$

$\underline{fact}\ ::=\ \ \textbf{\texttt{fact}}\ \ name\ block$

$\underline{cmd}\ ::=\ \big[\ name\ \big]\ cmdType\ \big(\ name\ \big|\ block\ \big)\ scope?$

$cmdType\ ::=\ \textbf{\texttt{run}}\ \big|\ \textbf{\texttt{check}}$

$\underline{scope?}\ ::=\ \big[\ \textbf{\texttt{for}}\ \ wholeNum\ \big|\ \textbf{\texttt{for}}\ \ wholeNum\ \textbf{\texttt{but}}\ \ typescopes\ \big|\ \textbf{\texttt{for}}\ \ typescopes\ \big]$

$typescopes\ ::=\ typescope\ \big(\ ,\ typescope\ \big)*$

$\underline{typescope}\ ::=\ \big[\ \textbf{\texttt{exactly}}\ \big]\ wholeNum\ name$

$\underline{term}\ ::=\ \big(\ expr\ \big|\ fmla\ \big)$

$\underline{fmla}\ ::=\ block\ \big|\ app\ \big|\ multFmla\ \big|\ quantFmla\ \big|\ logFmla\ \big|\ cmpFmla\ \big|\ cardFmla\ \big|\ letFmla$

$\underline{block}\ ::=\ \{\ fmla\star\ \}$

$exprMult\ ::=\ \textbf{\texttt{no}}\ \big|\ \textbf{\texttt{lone}}\ \big|\ \textbf{\texttt{one}}\ \big|\ \textbf{\texttt{some}}$

```
quantFmla ::= quant quantDecls block | quant quantDecls | fmla
quant ::= ( no | lone | one | some )
logFmla ::= not fmla | fmla binaryFmlaOp fmla | iteFmla
binaryFmlaOp ::= && | and | || | or | => | implies | <=> | iff
iteFmla ::= fmla implies fmla else fmla | fmla => fmla else fmla
cmpFmla ::= fmla [ not ] in fmla | fmla [ not ] = fmla
cardFmla ::= # name <= natNum | # name = natNum
letFmla ::= let letBindings block | let letBindings | fmla
not ::= ! | not

expr ::= atom | app | setComp | iteExpr | ~ expr | expr exprOp expr
atom ::= univ | none | iden | name
setComp ::= { quantDecls block } | { quantDecls | fmla }
iteFmla ::= fmla implies expr else expr | fmla => expr else expr
exprOp ::= + | - | & | . | <: | :> | ++

app ::= expr . expr | expr[expr ( , expr )*]

letBindings ::= letBinding ( , letBinding )*
letBinding ::= name = term

decls ::= decl ( , decl )*
decl ::= [ disj ] names : [ relMult ] expr

quantDecls ::= quantDecl ( , quantDecl )*
quantDecl ::= [ disj ] name ( , name )* : expr
```

names `::=` *name* (`,` *name*)∗

<u>*name*</u> `::=` (`A-Za-z`)(`A-Za-z0-9_'"`)∗

<u>*natNum*</u> `::=` (`0-9`)(`0-9`)∗

3.1.2 Notation (Elements of Supported Alloy)**.** The names of some rules in the above grammar are underlined. We highlight these rules because the syntactic expressions that follow them form sets that we refer to throughout this document. We denote these sets using cursive letters with the subscript "A" (for "Alloy"). The notation scheme is defined in Table A.

3.1.3 Notation (Alloy Code Within This Report)**.** We use the following conventions when writing Alloy code within this report:

- When a line of Supported Alloy code is presented within prose or mathematical expressions, it is enclosed in angle brackets. For example: ⟨**run** f **for** 2⟩.
- When a block of Supported Alloy code is presented, it is indented. For example:

```
pred f[x: A] {
    some x
    x in B
}
```

- Optional code phrases are enclosed in large square brackets. For example, ⟨**all** [**disj**] x, y:Z {}⟩ indicates that **disj** is optional. These brackets are not to be confused with literal brackets, `[` and `]`.
- Keywords and operators are written in bold monospace text. For example: **sig** , **implies** , **+** , **<=>**

- Literal symbols are written in plain monospace text. For example: `x`, `_hello_`.
- Lastly, variable terms (i.e., expressions, names, and symbols that depend on rules within this report) are written in italicized "math" text. For example: u, $CrRelCnstr(n)$.

3.2 Core Alloy

Core Alloy is a subset of Supported Alloy that is desugared, but also transformed in several key ways that prepare for the translation from Core Alloy into the Intermediate Language (IL). Core Alloy is still valid Alloy, and can be interpreted by the Alloy Analyzer.

The additional Alloy sig (**`sig`**) `_univ` is present in all Core Alloy files, and every other sig's entire declaration is of the form **`sig`** `n` **`in`** `_univ`, where `n` is the name of the sig. All relation declarations are made in the `_univ` sig. In Core Alloy, the inheritance hierarchy, as well as the types and properties of the declared relations, are modeled using **fact**s instead of Alloy's special syntactic constructs.

Some complicated terms are not allowed in favor of simpler alternatives. For example, **`let`** is not present in Core Alloy, box join is used only for function application, and dot join is only used for joining between relations.

Finally, scopes (e.g., **`for`** `3` **`but`** `1 S`, in the context of a **`run`** or **`check`**) are not allowed on the command. Instead, a scope is enforced using facts that make equivalent assertions about the cardinality of the relevant sigs. Additionally, there is an option (see 4.1.21) that, when enabled, makes it so the output Core Alloy explicitly enforces the implicit cardinality bound of 3 on all top-level sigs.

The translation we specify in Chapter 4 from Core Alloy to the Intermediate Language is defined for every valid Core Alloy file. This is in contrast to Alloy, which we do not translate some parts of, e.g. transitive closure.

The grammar of Core Alloy is defined below. Note that some of the rule names are underlined; this is explained in 3.2.2.

Definition 3.2.1 (Grammar of Core Alloy)**.**

$\underline{spec}\ \texttt{::=}\ vocab\ \ funDef{*}\ \ predDef{*}\ \ fact^{\star}\ \ cmd$

$\underline{vocab}\ \texttt{::=}\ univSigDecl\ \ subSigDecls$

$univSigDecl\ \texttt{::=}\ \texttt{\textbf{sig}}\ \texttt{\_univ \{}\ decls\ \texttt{\}}$

$subSigDecls\ \texttt{::=}\ \texttt{\textbf{sig}}\ \ name\ \big(\ \texttt{,}\ \ name\ \big){*}\ \texttt{\}}\ \texttt{\textbf{in}}\ \ \texttt{\_univ}$

$\underline{predDef}\ \texttt{::=}\ \texttt{\textbf{pred}}\ \ name\texttt{[}decls\texttt{]}\ \ block$

$\underline{funDef}\ \texttt{::=}\ \texttt{\textbf{fun}}\ \ name\texttt{[}decls\texttt{]}\ \texttt{:}\ \texttt{\_univ \{}\ expr\ \texttt{\}}$

$\underline{fact}\ \texttt{::=}\ \ \texttt{\textbf{fact}}\ \ block$

$\underline{cmd}\ \texttt{::=}\ cmdType\ \big(\ name\ \big|\ block\ \big)\ \texttt{\textbf{for}}\ \ z\ \texttt{\_univ}$

$cmdType\ \texttt{::=}\ \texttt{\textbf{run}}\ \big|\ \texttt{\textbf{check}}$

$\underline{fmla}\ \texttt{::=}\ block\ \big|\ app\ \big|\ multFmla\ \big|\ quantFmla\ \big|\ logFmla\ \big|\ cmpFmla\ \big|\ cardFmla$

$\underline{block}\ \texttt{::=}\ \texttt{\{}\ \ fmla^{\star}\ \ \texttt{\}}$

$exprMult\ \texttt{::=}\ \texttt{\textbf{no}}\ \big|\ \texttt{\textbf{lone}}\ \big|\ \texttt{\textbf{one}}\ \big|\ \texttt{\textbf{some}}$

$quantFmla\ \texttt{::=}\ quant\ \ decls\ \ block$

$quant\ \texttt{::=}\ \big(\ \texttt{\textbf{no}}\ \big|\ \texttt{\textbf{lone}}\ \big|\ \texttt{\textbf{one}}\ \big|\ \texttt{\textbf{some}}\ \big)$

$logFmla\ \texttt{::=}\ \texttt{\textbf{not}}\ \ fmla\ \big|\ fmla\ \ binaryFmlaOp\ \ fmla\ \big|\ iteFmla$

$binaryFmlaOp\ \texttt{::=}\ \texttt{\textbf{\&\&}}\ \big|\ \texttt{\textbf{||}}\ \big|\ \texttt{\textbf{=>}}\ \big|\ \texttt{\textbf{<=>}}$

$iteFmla\ \texttt{::=}\ fmla\ \texttt{\textbf{=>}}\ \ fmla\ \texttt{\textbf{else}}\ \ fmla$

$cmpFmla\ \texttt{::=}\ fmla\ \texttt{\textbf{in}}\ \ fmla\ \big|\ fmla\ \texttt{=}\ fmla$

$cardFmla\ \texttt{::=}\ \texttt{\textbf{\#}}\ name\ \texttt{\textbf{<=}}\ \ natNum\ \big|\ \texttt{\textbf{\#}}\ name\ \texttt{\textbf{=}}\ \ natNum$

$\underline{expr}$ `::=` $atom$ | app | $setComp$ | $iteExpr$ | **`~`** $expr$ | $expr$ $exprOp$ $expr$

$atom$ `::=` **`univ`** | **`none`** | **`iden`** | $name$

$setComp$ `::=` `{` $decls$ $block$ `}`

$iteFmla$ `::=` $fmla$ **`implies`** $expr$ **`else`** $expr$

$exprOp$ `::=` **`+`** | **`-`** | **`&`** | **`.`** | **`<:`** | **`:>`** | **`++`**

app `::=` $expr$`[`$expr$ (`,` $expr$)`*]`

$decls$ `::=` $decl$ (`,` $decl$)`*`

$\underline{decl}$ `::=` $name$ `:` `_univ`

$\underline{name}$ `::=` (`A-Za-z`)(`A-Za-z0-9_'"`)`*`

$\underline{natNum}$ `::=` (`0-9`)(`0-9`)`*`

3.2.2 Notation (Elements of Core Alloy)**.** The names of some rules in the above grammar are underlined. We highlight these rules because the syntactic expressions that follow them form sets that we refer to throughout this document. We denote these sets using cursive letters with the subscript “C” (for “Core”). The notation scheme is defined in Table A.

3.2.3 Notation (Core Alloy Code Within This Report)**.** Core Alloy code within this report is written using the same notation described in 3.1.3.

3.2.4 Notation (Products of `_univ`)**.**

$$\texttt{\_univ}^n = \langle \texttt{\_univ -> \_univ -> \_univ -> ...} \rangle \text{ for } n \text{ repetitions of “\_univ”.}$$

3.3 The Intermediate Language

The Intermediate Language (IL) is a higher-order, simply-typed lambda-calculus that is an abstraction over both Alloy and SMT, enabling us to bridge the gap between the two languages.

An Alloy programmer may think of Alloy expressions as sets of values, where each set contains values of the same arbitrary arity. For the purposes of this translation, though, we think of Alloy expressions as *relations*, where some relation e contains the mapping $a_1 \to a_2 \to \cdots \to a_N$ if and only if `a1 -> a2 -> ...-> aN in e`. The Intermediate Language stores relations as single-argument predicates, (i.e., Boolean-valued functions), where the argument is a tuple with the same arity as the relation.

The IL has one sort, `univ`, because we use unary predicates to represent Alloy sigs, which itself is because SMT-LIB sorts are disjoint, while Alloy sorts can have complex inheritance patterns. See section 2.1 for a thorough discussion of why we represent Alloy sigs as unary predicates.

As an example illustrating the above descriptions, given an Alloy expression e with the type `A -> B -> C`, the type of $TrExpr(e)$ is $U \to U \to U \to \mathbb{B}$.

Definition 3.3.1 (IL Theories)**.** An IL theory is an ordered pair $< v, A >$ where:

- v is the signature of the theory.
- A is a set of Boolean-typed terms that make up the axioms of the theory.

Definition 3.3.2 (IL Signatures)**.** An IL signature (not to be confused with an Alloy or Core Alloy sig, which has a completely different meaning), is a set of constant function declarations, each of which is denoted by $n : T \to V$ for some name n, 0-order IL type T, and atomic IL type V.

The IL is a simply typed lambda calculus with the following types:

Definition 3.3.3 (IL Types)**.**

1. **Atomic Types:** There are two atomic types, U and $\mathbb{B}$.

2. **Tuple Type:** A tuple type is the Cartesian product $T_1 \times T_2 \times \cdots \times T_k$, for $k \geq 0$, denoted $< T_1, \ldots, T_k >$.

3. **Function Type:** A function type is a parameter type T and a return type V, written $T \rightarrow V$.

3.3.4 Notation (Tuple Shorthand: T^n)**.** For convenience, we define T^n to be the tuple type $< T \ , \ T \ , \ \ldots \ , \ T >$ for n repetitions of T, where T is a type and $n > 0$.

Definition 3.3.5 (Order of Types)**.**

1. A type T has order 0 if it is either a base type or is of the form $(T_1 \times \cdots \times T_n)$, $n > 1$, with each T_i of order 0.

2. A type T has order 1 if it is $(T_1 \rightarrow \cdots \rightarrow T_n)$, $n > 1$, with each T_i of order 0.

We say that a term has order 0 (or 1) if its type has order 0 (or 1).

An IL term can be:

Definition 3.3.6 (IL Terms)**.** An IL term takes one of the following forms:

1. **Constant:** A constant term is an ordered pair $< n, t >$ of a name n and a type t. Declared functions are constants.

2. **Variable:** A variable term is an ordered pair $< n, t >$ of a name n and a type t.

3. **Tuple:** Given terms $t_1, \ldots, t_k$ ($k \geq 0$) whose types are $T_1, \ldots, T_k$, the term $< t_1, \ldots, t_k >$ is a tuple of type $< T_1, \ldots, T_k >$. The terms of a tuple do not need to be of the same type.

4. **Abstraction:** Given a variable x whose type is T, and a term v whose type is V, $\lambda x.v$ is a function of type $T \to V$.

5. **Application:** Given a term t of type $S \to V$ and a term s of type S, ts is a term of type V.

6. **Projection:** Given a term $< t_1, \ldots, t_k >$ of type $< T_1, \ldots, T_k >$, and an integer i, $\pi^i t$ is a term of type T_i.

7. **Forall:** Given a term b of Boolean type and a variable x which is free in b, $\forall x.b$ is a term of Boolean type.

8. **Exists:** Given a term b of Boolean type and a variable x which is free in b, $\exists x.b$ is a term of Boolean type.

While discussing the Intermediate Language we will use the following notation:

3.3.7 Notation (Substitution)**.** Use $t[x := s]$ to denote the usual capture-avoiding substitution of s for x in t.

Note that $t[y := r][x := s] = (t[x := s])[y := r]$

The terms of the Intermediate Language are subject to the following two axioms:

Definition 3.3.8 (Axioms)**.**

1. $\pi^i < t_1, \ldots, t_n >= t_i$, where t is a tuple whose ith element is t_i
2. $(\lambda x.v)u = v[x := u]$

Definition 3.3.9 (Reduction)**.**

1. $\pi^i < t_1, \ldots, t_n > \to t_i$, where t is a tuple whose ith element is t_i
2. $(\lambda x.v)u \to v[x := u]$

Definition 3.3.10. $t \twoheadrightarrow s$ if there is a sequence $\{t_i\}_1^n$ such that $t \to t_1 \to \cdots \to t_n \to s$, where $n \geq 0$.

3.3.1 Relationship to the Haskell Data Structure

The IL is motivated by the desire to represent higher order constructions, specifically abstraction, in a format reminiscent of SMT-LIB. The translator produces IL which, when normalized, contains no abstractions or higher order constructions, and so can be reduced into SMT-LIB.

In the Haskell data structure representing the IL, there are three operations related to tuples: `tuple`, which creates a tuple, `concat`, which takes two tuples and combines them into one (flat) tuple, and `slice`, which takes two numbers i and j and a tuple, and returns a tuple which is the elements of the given tuple whose index is between i and j, inclusive. This is fine programmatically, but it is a relatively non-standard way to represent a lambda calculus mathematically, so we chose to instead use the more standard tuple ($< t_1, \ldots, t_n >$) and projection ($\pi^i t$) constructors.

Similarly, in the Haskell data structure representing the IL, there is a single type of term `ref` which can model both constants and variables. This was useful programmatically, but it was more convenient mathematically to represent variables and constants with two separate kinds of terms.

3.4 SMT-LIB

SMT-LIB is a many-sorted logical specification language closely based in first-order logic. The building blocks of an SMT-LIB specification are terms, which are logical statements. Basic boolean operations like `and` and `or`, and quantification (`forall` and `exists`) are examples of terms.

There are three top level syntactic constructs that contribute to the specification of an SMT-LIB theory[1]: sort declarations, function definitions and declarations, and assertions.

[1]Most implementations of SMT-LIB have a few additional constructs that can control an SMT solver's

Sort declarations declare sorts, which are sets. All sorts are disjoint, and SMT-LIB terms can reference sorts. SMT-LIB supports declaration of functions (specifying only their types) and definition of functions, where the user provides a specific value for the function. Finally, an assertion simply wraps a boolean-valued term, and commands the SMT solver to only search for models where the term's value is `True`.

For a mathematically precise definition of SMT-LIB and the SMT-LIB grammar, see [BFT17], the SMT-LIB language standard.

behavior. For example, the Z3 analyzer accepts the command `(check-sat`, which causes it to print either "sat" or "unsat" depending on the given specification

Chapter 4

The Translation

Recall that the translation is conducted in six steps:

- **Lexing**, in which the text of the given Alloy specification is tokenized.
- **Parsing**, in which the tokenized Alloy is parsed into an abstract syntax tree.
- **Translation to Core**, in which the Alloy abstract syntax tree is desugared and simplified into a Core Alloy syntax tree.
- **Translation to IL**, in which the Core Alloy syntax tree is lifted into higher order logic and represented in our Intermediate Language (IL).
- **Normalization**, in which the IL terms from the translation to IL are beta reduced. For input specifications generated by the translation to IL, the resulting specifications never contain abstraction terms.
- **Translation to SMT-LIB**, in which the IL terms are expressed in SMT-LIB. Mathematically, this step is simple, but programmatically it involves converting tuples and variables of tuple type into lists, and converts from IL data structures to SMT-LIB data structures.

In this chapter, we give a detailed description of the translation to core, translation to IL, and a brief description of normalization and translation to SMT-LIB.

4.1 Supported Alloy to Core Alloy

The name of Core Alloy implies that the translation to it should simply desugar Alloy. The translation to core does desugar the given Alloy, but it also transforms the Alloy in other ways to prepare for the next step, translation into the Intermediate Language. The translation to core removes many complex syntactic constructs like relation types and the inheritance hierarchy between Alloy sigs, and expresses these with facts instead. By simplifying the syntax and structure of the Alloy specification, the translation to core allows the translation to the Intermediate Language to be simpler and more focused.

4.1.1 Notation (Translation Functions)**.** In this section, we describe the translation to Core Alloy using a number of mutually-recursive functions:

- *CrSpec*: Translate from an Alloy specification to a Core Alloy specification.
- *CrVoc*: Translate from an Alloy vocabulary to a Core Alloy vocabulary.
- *CrSigInhCnstr*: Extract a Core Alloy constraint that replicates the properties of the inheritance hierarchy described by an Alloy sig extension clause.
- *SigChildren*, *SigParents*, *SigDisjSiblings*: Get the children of, parents of, and siblings that are disjoint from the given sig (respectively).
- *CrDecl*: Translate from an Alloy declaration to a Core Alloy declaration.
- *CrRelCnstr*: Extract Core Alloy constraints enforcing the type, multiplicity, and (if applicable) disjointness of relations declared in Alloy.
- *CrParamCnstr*: Extract Core Alloy constraints enforcing the type, multiplicity, and (if applicable) disjointness of predicate parameters declared in Alloy.
- *CrDeclDisjCnstr*: If applicable, extract a Core Alloy constraint enforcing disjointness between elements declared in Alloy. Otherwise, return a tautology.

- *CrPred*: Translate from an Alloy predicate definition to a Core Alloy predicate definition.
- *CrFunc*: Translate from an Alloy function definition to a Core Alloy function definition.
- *CrAssert*: Translate from an Alloy assertion to a Core Alloy predicate definition.
- *CrFact*: Translate from an Alloy fact to a Core Alloy fact.
- *CrCmd*: Translate from an Alloy command to a Core Alloy command.
- *CrPredRun*: Translate from a directly-run Alloy predicate to a directly-run Core Alloy predicate.
- *CrScope*: Translate from an Alloy scope to a Core Alloy formula block replicating the constraints imposed by that scope.
- *CrTScope*: Translate from an Alloy type-scope to a Core Alloy formula block replicating the constraints imposed by that type-scope.
- *CrFmla*: Translate from an Alloy formula to a Core Alloy formula.
- *CrQFmla*: Translate from an Alloy quantifier formula to a Core Alloy quantifier formula.
- *CrExpr*: Translate from an Alloy expression to a Core Alloy expression.
- *CrSetCompr*: Translate from an Alloy set comprehension to a Core Alloy set comprehension.
- *CrJoin*: Translate from an Alloy dot- or box- join to a Core Alloy dot- or box-join.

Definition 4.1.2 (Translation of Specifications: *CrSpec*)**.** Let t be a Supported Alloy specification consisting of v, P, U, R, F, and c, where:

- $v \in \mathcal{V}_A$ is the vocabulary of t, i.e. the set of sig and relation declarations in t.
- $P \subseteq \mathcal{P}_A$ is the set of predicate definitions in t.
- $U \subseteq \mathcal{U}_A$ is the set of function definitions in t.
- $R \subseteq \mathcal{R}_A$ is the set of assertions in t.
- $F \subseteq \mathcal{B}_A$ is the set of blocks that make up the facts of t.
- $c \in \mathcal{C}_A$ is the command[1] in t.

[1]Although Alloy specifications may have more than one command, Supported Alloy only includes specifications with a single command.

Then, the translation $\mathit{CrSpec}(t)$ of t contains v', P', U', F', and c, where:

- $v' = \mathit{CrVoc}(v)$ is the translated vocabulary.
- $P' = \{\mathit{CrPred}(p) \mid p \in P\}$ is the set of translated predicate definitions.
- $U' = \{\mathit{CrFunc}(u) \mid u \in U\}$ is the set of translated function definitions.
- $F' = \{\mathit{CrFmla}(f) \mid f \in F\}$ is the set of translated fact blocks.
- $c' = \mathit{CrCmd}(c)$ is the translated command, which includes exactly one Core Alloy command, and zero or one Core Alloy predicate definitions.

Definition 4.1.3 (Translation of Vocabularies: *CrVoc*)**.** Let $v =$

$$\left[\ \textbf{abstract}\ \right]\ m_1\ \textbf{sig}\ n_{1,1}\ \ldots\ n_{1,k_1}\ x_1\ \{\ d_{1,1}\ \ldots\ d_{1,\ell_1}\ \}$$

$$\ldots$$

$$\left[\ \textbf{abstract}\ \right]\ m_j\ \textbf{sig}\ n_{j,1}\ \ldots\ n_{j,k_j}\ x_n\ \{\ d_{j,1}\ \ldots\ d_{j,\ell_j}\ \}$$

be an Alloy vocabulary[2], where, for all $i > 0$:

- $m_i \in \{\textbf{lone}, \textbf{one}, \textbf{some}\}$ is a “sig multiplicity”.
- $n_{i,1}$, … , n_{i,k_i} are “sig names”.
- x_i is a sig extension clause.
- $d_{i,1}$, … , d_{i,ℓ_i} are “relation declarations”.

The translation $\mathit{CrVoc}(v)$ of v is given by:

```
// Universal sig with all relations
sig _univ {
    $CrDecl(d_{1,1})$ , … , $CrDecl(d_{1,\ell_1})$ ,
    … ,
    $CrDecl(d_{j,1})$ , … , $CrDecl(d_{j,\ell_j})$
}
// Sub-sigs, all in universal sig
sig $n_{1,1}$ , … , $n_{1,k_1}$
```

[2]Note that for all i, it is also syntactically valid for **abstract** to appear directly after m_i, which has the same effect as it appearing before m_i.

```
        ... ,
        $n_{j,1}$ , ... , $n_{j,k_j}$ in _univ {}
    // Fact replicating sig multiplicities
    fact {
        ($m_1$ $n_{1,1}$) && ... && ($m_1$ $n_{1,k_1}$)
        ...
        ($m_j$ $n_{j,1}$) && ... && ($m_j$ $n_{j,k_j}$)
    }
    // Fact replicating inheritance relationships
    fact {
        $CrSigInhCnstr(n_{1,1})$ && ... && $CrSigInhCnstr(n_{1,k_1})$
        ...
        $CrSigInhCnstr(n_{j,1})$ && ... && $CrSigInhCnstr(n_{j,k_j})$
    }
    // Fact replicating declaration types and multiplicities
    fact {
        $CrRelCnstr(n_1, d_{1,1})$ && ... && $CrRelCnstr(n_1, d_{1,\ell_1})$
        ...
        $CrRelCnstr(n_j, d_{j,1})$ && ... && $CrRelCnstr(n_j, d_{j,\ell_j})$
    }
```

4.1.4 Example (Translation of a Vocabulary)**.** Let $v =$

```
    some sig A { }
    sig B { r: A -> A }
```

The translation $CrVoc(v)$ of v is then:

```
    // Universal sig with all relations
```

```
sig _univ { r: _univ -> _univ }
// Sub-sigs, all in universal sig
sig A, B in _univ {}
// Fact replicating sig multiplicities
fact { some A }
// Fact replicating inheritance relationships
fact { no (A & B) }
// Fact replicating declaration types and multiplicities
fact { r in (B -> A -> A) }
```

Definition 4.1.5 (Extracting Signature Inheritance Constraints: *CrSigInhCnstr*)**.** Let $n \in \mathcal{N}$ be a given sig name. We define the inheritance constraint $\mathit{CrSigInhCnstr}(n)$ of the sig whose name is n as $\langle f_p \ \texttt{\&\&} \ f_d \ \texttt{\&\&} \ f_a \rangle$, where:

- $f_p = \langle \texttt{(}n \ \textbf{in} \ \texttt{(}p_1 \ \texttt{+} \ldots \ \texttt{+} \ p_k\texttt{))} \rangle$ constrains the sig to be a subset of its parents, where $\{p_1 , \ldots , p_k\} = \mathit{SigParents}(n)$
- $f_d = \langle \textbf{no} \ \texttt{((}n \ \texttt{\&} \ b_1\texttt{)} \ \texttt{+} \ldots \ \texttt{+} \ \texttt{(}n \ \texttt{\&} \ b_j\texttt{)))} \rangle$ constrains the sig to be disjoint with its siblings, where $\{b_1 , \ldots , b_j\} = \mathit{SigDisjSiblings}(n)$. However, if $\mathit{SigDisjSiblings}(n)$ is an empty set [3], then $f_d = \langle \texttt{\{\}} \rangle$.
- f_a constrains an abstract sig to be a subset of its children, and is defined as:

$$f_a = \begin{cases} \langle \textbf{no} \ n \rangle & \text{if } k = 0 \\ \langle n \ \textbf{in} \ \texttt{(}c_1 \ \texttt{+} \ldots \ \texttt{+} \ c_k\texttt{)} \rangle & \text{if } n \text{ is declared with } \textbf{abstract} \\ \langle \texttt{\{\}} \rangle & \text{otherwise} \end{cases}$$

[3] $\mathit{SigDisjSiblings}(n)$ is an empty set when n is **in** its parents or n has no siblings.

where $\{c_1\ ,\ \ldots\ ,\ c_k\} = \mathit{SigChildren}(n)$.

4.1.6 Example (Extracting a Signature Inheritance Constraint)**.** Consider the following sig declarations.

```
abstract sig A {}
sig B, C in A {}
```

Then, $\mathit{CrSigInhCnstr}(A) = \Big\langle$`(A in _univ) && ({}) && (A in (B + C))`$\Big\rangle$.

And, $\mathit{CrSigInhCnstr}(B) = \Big\langle$`(B in _univ) && (no (B & C)) && ({})`$\Big\rangle$.

Definition 4.1.7 (Signature Inheritance Inspection: *SigParents*, *SigChildren*, *SigDisjSiblings*)**.** Consider a Supported Alloy specification with a set of sig declarations S, whose names form the set N. Let $s \in S$ be a sig declaration with the name $n \in N$ and extension clause x. We define the following functions for n, which provide information about the relationship of s with other sig declarations in the inheritance hierarchy.

- $\mathit{SigParents}(n)$ gives the names of the parents of s, and is defined as:

$$\mathit{SigParents}(n) = \begin{cases} \{\texttt{\_univ}\} & \text{if } x = \big\langle\big\rangle \\ \{p\} & \text{if } x = \big\langle \texttt{\textbf{extends}}\ p \big\rangle \\ \{p_1, \ldots, p_k\} & \text{if } x = \big\langle \texttt{\textbf{in}}\ p_1\ \texttt{\textbf{+}} \ldots\ \texttt{\textbf{+}}\ p_k \big\rangle \end{cases}$$

- $\mathit{SigChildren}(n)$ gives the names of the children of s, and is defined as:

$$\mathit{SigChildren}(n) = \{c \in N \mid c\ \texttt{\textbf{extends}} \text{ or is } \texttt{\textbf{in}}\ n\}$$

- $\mathit{SigDisjSiblings}(n)$ gives the *only* the siblings of s with which it must be disjoint, and is defined as:

$$SigDisjSiblings(n) = \begin{cases} \{\} & \text{if } x = \langle \texttt{\textbf{in}} \ldots \rangle \\ \{b \in N \mid (SigParents(n) \cap SigParents(b)) \neq \{\}\} & \text{otherwise} \end{cases}$$

Definition 4.1.8 (Translation of Declarations: *CrDecl*)**.** Let $d =$

$$\left[\ \texttt{\textbf{disj}}\ \right]\ n_1\ \ldots\ n_h\ \texttt{:}\ \left[\ m\ \right]\ e$$

be an Alloy declaration, where:

- n_1 , ... , $n_h \in \mathcal{N}$ are the "declared names".
- $m \in \{\texttt{\textbf{lone}}, \texttt{\textbf{one}}, \texttt{\textbf{set}}, \texttt{\textbf{some}}\}$ is the "declared multiplicity", which is not used in this rule.
- $e \in \mathcal{E}_A$ is the "declared type".

The translation $CrDecl(d)$ of d is given by:

$$n_1\ \texttt{:}\ \texttt{\_univ}^{Ar(e)}\ \texttt{,}\ \ldots\ \texttt{,}\ n_h\ \texttt{:}\ \texttt{\_univ}^{Ar(e)}$$

4.1.9 Example (Translating a Declaration)**.** Let $d = \langle$`r, s : T -> V`$\rangle$.

Then, $CrDecl(d) = \langle$`r : _univ -> _univ, s : _univ -> _univ`$\rangle$.

Definition 4.1.10 (Extracting Constraints from Relation Declarations: *CrRelCnstr*)**.** Let s be a sig name.

Let $d =$

$$\left[\ \texttt{\textbf{disj}}\ \right]\ n_1\ \ldots\ n_h\ \texttt{:}\ \left[\ m\ \right]\ e$$

be an Alloy relation declaration on s, where:

- n_1 , ... , $n_h \in \mathcal{N}$ are the "declared names".
- $m \in \{\texttt{\textbf{lone}}, \texttt{\textbf{one}}, \texttt{\textbf{set}}, \texttt{\textbf{some}}\}$ is the "declared multiplicity", and defaults to **`set`** if not given.
- $e \in \mathcal{E}_A$ is the "declared type".

Then, the constraints $CrRelCnstr(s,d)$ extracted from a relation d declared in sig s are:

$(n_1$ **+** $\ldots$ **+** $n_h)$ **in** $(s$ **->** $e)$

all _var_s: _univ {

(_var_s **in** s) **=>** (

($m\prime$ (_var_s. n_1)) ... ($m\prime$ (_var_s. n_h))

)

}

$CrDeclDisjCnstr(d)$

where:

$$m' = \begin{cases} \langle m \rangle & \text{if } m \in \{\texttt{lone}, \texttt{one}, \texttt{some}\} \\ \langle\rangle & \text{if } m = \texttt{set} \end{cases}$$

Definition 4.1.11 (Extracting Constraints from Parameter Declarations: *CrParamCnstr*)**.**

Let $d =$

$[$ **disj** $]$ n_1 $\ldots$ n_h **:** $[$ m $]$ e

be an Alloy declaration, where:

- n_1 , $\ldots$, $n_h \in \mathcal{N}$ are the "declared names".
- $m \in \{\texttt{lone}, \texttt{one}, \texttt{set}, \texttt{some}\}$ is the "declared multiplicity", and defaults to **set** if not given.
- $e \in \mathcal{E}_A$ is the "declared type".

Then, the constraints $CrParamCnstr(d)$ extracted from d are:

$(n_1$ **+** $\ldots$ **+** $n_h)$ **in** e

$(m\prime$ $n_1)$ $\ldots$ $(m\prime$ $n_h)$

$CrDeclDisjCnstr(d)$

where:

$$m' = \begin{cases} \langle m \rangle & \text{if } m \in \{\texttt{lone}, \texttt{one}, \texttt{set}, \texttt{some}\} \\ \langle\rangle & \text{if } m = \texttt{set} \end{cases}$$

Definition 4.1.12 (Extracting Disjointness Constraints from Declarations: *CrDeclDisjCnstr*)**.**

Let $d =$

$$\left[\ \texttt{disj}\ \right]\ n_1\ \ldots\ n_h\ :\ \left[\ m\ \right]\ e$$

be an Alloy declaration, where:

- $n_1\ ,\ \ldots\ ,\ n_h \in \mathcal{N}$ are the "declared names".
- $m \in \{\texttt{lone}, \texttt{one}, \texttt{set}, \texttt{some}\}$ is the "declared multiplicity", which is not used

  in this rule.
- $e \in \mathcal{E}_A$ is the "declared type".

If **`disj`** is present in the declaration, then the extracted disjointness constraint $CrDeclDisjCnstr(d)$

=

```
no  (
     (n_1 & n_2) + (n_1 & n_3) + ... + (n_1 & n_h) +
     (n_2 & n_3) + (n_2 & n_4) + ... + (n_2 & n_h) +
     ... +
     (n_{h-2} & n_h) + (n_{h-1} & n_h) +
     (n_{h-1} & n_h)
)
```

Otherwise, $CrDeclDisjCnstr(d) = \langle \{\} \rangle$.

4.1.13 Example (Extracting a Disjointness Constraint)**.** Consider the Supported Alloy declaration $\langle$**`disj`** `q, r, s, t: A`$\rangle$.

Its extracted disjointness constraint would be:

```
no  (
     (q & r) + (q & s) + (q & t) +
     (r & s) + (r & t) +
     (s & t)
)
```

Definition 4.1.14 (Translation of Predicate Definitions: *CrPred*)**.** Let p, an Alloy predicate definition, equal one of the following two (semantically-equivalent) phrases:

- $\left\langle \texttt{\textbf{pred}}\, n\texttt{[}d_1\ \texttt{,}\ \ldots\ \texttt{,}\ d_h\texttt{]}\ \ b \right\rangle$
- $\left\langle \texttt{\textbf{pred}}\, n\texttt{(}d_1\ \texttt{,}\ \ldots\ \texttt{,}\ d_h\texttt{)}\ \ b \right\rangle$

where:

- $n \in \mathcal{N}$ is the name of the predicate.
- $d_1\ ,\ \ldots\ ,\ d_h \in \mathcal{D}_A$ are the “parameter declarations”.
- $b \in \mathcal{B}_A$ is a block that is the “body”.

Then, the translation $CrPred(p)$ of p is given by:

$$\texttt{\textbf{pred}}\ \ n\texttt{[}CrDecl(d_1)\ \texttt{,}\ \ldots\ \texttt{,}\ CrDecl(d_h)\texttt{]}\ \ CrFmla(b)$$

Definition 4.1.15 (Translation of Function Definitions: *CrFunc*)**.** Let u, an Alloy function definition, equal one of the following two (semantically-equivalent) phrases:

- $\left\langle \texttt{\textbf{fun}}\, n\texttt{[}d_1\ \texttt{,}\ \ldots\ \texttt{,}\ d_h\texttt{]}\ \texttt{:}\ \ e_1\ \texttt{\{}\ e_2\ \texttt{\}} \right\rangle$
- $\left\langle \texttt{\textbf{fun}}\, n\texttt{(}d_1\ \texttt{,}\ \ldots\ \texttt{,}\ d_h\texttt{)}\ \texttt{:}\ \ e_1\ \texttt{\{}\ e_2\ \texttt{\}} \right\rangle$

where:

- $n \in \mathcal{N}$ is the name of the function.
- $d_1\ ,\ \ldots\ ,\ d_h \in \mathcal{D}_A$ are the “parameter declarations”.
- $e_1 \in \mathcal{E}_A$ is the “return type”.
- $e_2 \in \mathcal{E}_A$ is the “return value”.

Then, the translation $CrFunc(u)$ of u is given by:

$$\texttt{\textbf{fun}}\ \ n\texttt{[}CrDecl(d_1)\ \texttt{,}\ \ldots\ \texttt{,}\ CrDecl(d_h)\texttt{]}\ \texttt{:}\ \texttt{\_univ}^{Ar(e_1)}\ \texttt{\{}\ CrExpr(e_2)\ \texttt{\}}$$

Definition 4.1.16 (Translation of Assertions: *CrAssert*)**.** Let r, an Alloy assertion, equal one of the following:

- $\langle$**`assert`** $n\ \ b\,\rangle$
- $\langle$**`assert`** $b\,\rangle$

where:

- $n \in \mathcal{N}$ is the "name".
- $b \in \mathcal{B}_A$ is a block that is the "body".

If n is not specified, then the entire assertion is ignored, as there is no way to reference an unnamed assertion in Alloy. Otherwise, the translation $\mathit{CrAssert}(r)$ of r is given by:

`pred` `_assert_`n`[]` $\mathit{CrFmla}(b)$

Definition 4.1.17 (Translation of Facts: *CrFact*)**.** Let a, an Alloy fact, equal one of the following:

- $\langle$**`fact`** $n\ \ b\,\rangle$
- $\langle$**`fact`** $b\,\rangle$

be an Alloy fact, where:

- $n \in \mathcal{N}$ is an optional "name", which we ignore.
- $b \in \mathcal{B}_A$ is a block that is the "body".

Then, the translation $\mathit{CrFact}(a)$ of a is given by:

`fact` $\mathit{CrFmla}(b)$

Definition 4.1.18 (Translation of Commands: *CrCmd*)**.** Given an Alloy command $c \in \mathcal{C}_A$, its translation $\mathit{CrCmd}(g)$ is defined as:

$$CrCmd(c) = \begin{cases} \langle \texttt{\textbf{run}} \ \{ \ CrScope(k) \ \texttt{\textbf{\&\&}} \ \texttt{\textbf{!}} \ CrFmla(b) \ \} \ \texttt{\textbf{for}} \ z \ \texttt{\_univ} \rangle & \text{if } g = \langle \texttt{\textbf{check}} \ b \ \ k \rangle; b \in \mathcal{B}_A, k \in \mathcal{K}_A \\ \langle \texttt{\textbf{run}} \ \{ \ CrScope(k) \ \texttt{\textbf{\&\&}} \ \texttt{\textbf{!}} \ \texttt{\_assert\_}n \ \} \ \texttt{\textbf{for}} \ z \ \texttt{\_univ} \rangle & \text{if } g = \langle \texttt{\textbf{check}} \ n \ \ k \rangle; n \in \mathcal{N}, k \in \mathcal{K}_A \\ \langle \texttt{\textbf{run}} \ \{ \ CrScope(k) \ \texttt{\textbf{\&\&}} \ CrFmla(b) \ \} \ \texttt{\textbf{for}} \ z \ \texttt{\_univ} \rangle & \text{if } g = \langle \texttt{\textbf{run}} \ b \ \ k \rangle; b \in \mathcal{B}_A, k \in \mathcal{K}_A \\ CrPredRun(g) & \text{if } g = \langle \texttt{\textbf{run}} \ n \ \ldots \rangle; n \in \mathcal{N}, k \in \mathcal{K}_A \end{cases}$$

where z is the sum of the upper cardinality bounds of all top-level sigs in the original Alloy specification.

Note that $\langle \texttt{\textbf{for}} \ z \ \texttt{\_univ} \rangle$ is only included so that the Core Alloy specification is semantically equivalent to the source Alloy specification; it is disregarded when translating to the IL. We do not elaborate any more on this specific subtlety, as it is not a mathematically interesting problem in the context of this project.

Definition 4.1.19 (Translation of Directly-Run Predicates: *CrPredRun*)**.** Let $c = \langle \texttt{\textbf{run}} \ n \ \ k \rangle$ be an Alloy command that directly runs a predicate. Let p, the Alloy predicate named by n, be declared in one of the following (semantically-equivalent) ways:

- $\langle \texttt{\textbf{pred}} \ n \, [d_1 \ , \ \ldots \ , \ d_h] \ \ b \ \rangle$
- $\langle \texttt{\textbf{pred}} \ n \, (d_1 \ , \ \ldots \ , \ d_h) \ \ b \ \rangle$

where:

- $n \in \mathcal{N}$ is the name of the predicate.
- $d_1 \ , \ \ldots \ , \ d_h \in \mathcal{D}_A$ are the "parameter declarations".
- $b \in \mathcal{B}_A$ is a block that is the "body".

Then, the translation $CrPredRun(c)$ of c is given by:

$\texttt{\textbf{pred}} \ \ \texttt{\_proxy\_}n[d\prime_1 \ , \ \ldots \ , \ d\prime_k] \ \{$

$\quad CrParamCnstr(d_1) \ \ldots \ CrParamCnstr(d_h)$

$\quad CrScope(k)$

```
        p[a_1 , ... , a_k]
    }
    run _proxy_n for z _univ
```

where:

- $d'_1 \ldots d'_k = \mathit{CrDecl}(d_1)\ldots\mathit{CrDecl}(d_h)$
- $a_1 \ldots a_k$ are the names from the declarations $d'_1 \ldots d'_k$.
- $z \in \mathbb{N}$ is the sum of the upper cardinality bounds of all top-level sigs in the original Alloy specification.

Definition 4.1.20 (Translation of Command Scopes: *CrScope*)**.** Given an Alloy command scope k, define its translation $\mathit{CrScope}(k)$ as:

$$\mathit{CrScope}(k) = \begin{cases} \langle \texttt{\{\}} \rangle & \text{if } k = \langle\rangle \text{ and } \Theta = \bot \\ \langle \texttt{\{ (\#}\, s_1 \;\texttt{<=}\; 3\texttt{)} \ldots \texttt{(\#}\, s_j \;\texttt{<=}\; 3\texttt{) \}} \rangle & \text{if } k = \langle\rangle \text{ and } \Theta = \top \\ \langle \texttt{\{ (\#}\, s_1 \;\texttt{<=}\; z\texttt{)} \ldots \texttt{(\#}\, s_j \;\texttt{<=}\; z\texttt{) \}} \rangle & \text{if } k = \langle \texttt{for}\; z \rangle \\ \langle \texttt{\{}\; \mathit{CrTScope}(y_1)\ldots\mathit{CrTScope}(y_h)\;\texttt{(\#}\, s_1 \;\texttt{<=}\; z\texttt{)} \ldots \texttt{(\#}\, s_j \;\texttt{<=}\; z\texttt{) \}} \rangle & \text{if } k = \langle \texttt{for}\; z \;\texttt{but}\; y_1\texttt{,}\ldots\texttt{,}y_h \rangle \\ \langle \texttt{\{}\; \mathit{CrTScope}(y_1)\ldots\mathit{CrTScope}(y_h)\;\texttt{\}} \rangle & \text{if } k = \langle \texttt{for}\; y_1\texttt{,}\ldots\texttt{,}y_h \rangle \end{cases}$$

where:

- $z \in \mathbb{N}$ is the “general bound”.
- $y_1 , \ldots , y_h \in \mathcal{Y}_A$ are the “type-scopes”.
- $s_1 , \ldots , s_j \in \mathcal{N}$ are the Alloy top-level sigs **not** referenced in any of the type-scopes $y_1 , \ldots , y_h$.
- Θ is a boolean indicating whether the default cardinality bound (see 4.1.21) is being enforced.

4.1.21 Note (Default Cardinality Bound)**.** When searching for instances of a specification, Alloy places a default upper bound of 3 on the cardinalities of all top-level sigs that are not otherwise explicitly bounded. This is due to a technical restriction of the Alloy Analyzer that requires boundedness in order to analyze a specification.

SMT solvers do not have this technical restriction, so implicit bounds are not required. It is reasonable, then, to think that the translator should simply disregard this default cardinality bound. However, it can be argued that, in order for the translation to correctly preserve satisfiability, the translator *should* replicate the default cardinality bound.

We strike a compromise: the definition of *CrScope* (see 4.1.20) describes how to translate scopes in both cases.

Definition 4.1.22 (Translation of Command Type-Scopes: *CrTScope*)**.** Let $y =$

$$\Big[\ \textbf{exactly}\ \Big]\ z\ n$$

be an Alloy command type-scope, where:

- $z \in \mathbf{N}$ is the "upper bound".
- $n \in \mathcal{N}$ is the name of the sig for which the type-scope establishes a bound.

Then,

$$CrTScope(y) = \begin{cases} \langle \texttt{\#}\ n\ \texttt{=}\ z \rangle & \text{if } \textbf{exactly} \text{ is included} \\ \langle \texttt{\#}\ n\ \texttt{<=}\ z \rangle & \text{otherwise} \end{cases}$$

4.1.23 Example (Translating a Command that Directly Runs a Predicate)**.** Consider the following Supported Alloy specification.

```
sig A, B {}
pred p[a : one A] { ... }
run p for 5
```

The Core Alloy translation of the **run** command is:

```
pred _proxy_p[a : _univ] {
    (a in A) (one a) (#A <= 5) p[a]
}
run _proxy_p for 10
```

4.1.24 Example (Translating a Command with a Scope)**.** Consider a Supported Alloy specification with the zero-argument predicate `q`, sigs `A`, `B`, `C` and the command:

```
check  q  for 4 but 1 A, exactly 2 B
```

The Core Alloy translation of such a command is:

```
run {
    { (#A <= 1) (#B = 2) (#C <= 4) } &&
    !q
} for 7 _univ
```

4.1.25 Notation (Symbols Used in Term Translation)**.** Within the definitions of *CrExpr* and *CrFmla*:

- All instances of n represent an element of $\mathcal{N}$, i.e. a name.
- All instances of e or e_h (for $h > 0$) represent an element of $\mathcal{E}_A$, i.e. an Alloy expression.
- All instances of f or f_h (for $h > 0$) represent an element of $\mathcal{F}_A$, i.e. an Alloy formula.
- All instances of b or b_h (for $h > 0$) represent an element of $\mathcal{B}_A$, i.e. a formula block.
- All instances of m or m_h (for $h > 0$) represent an element of $(\mathcal{E}_A \cup \mathcal{F}_C)$, i.e. an Alloy formula or expression.
- All instances of q or q_h (for $h > 0$) represent an element of $\mathcal{Q}_A$, i.e. a quantifier variable declaration.

Definition 4.1.26 (Translation of Formulas: *CrFmla*)**.** Given an Alloy formula $g \in \mathcal{F}_A$, its translation $CrFmla(g)$ is defined as:

$$CrFmla(g) = \begin{cases}
\langle CrExpr(e_1)\ \texttt{in}\ CrExpr(e_2)\rangle & \text{if } g = \langle e_1\ \texttt{in}\ e_2\rangle \\
\langle \texttt{!}\ (CrExpr(e_1)\ \texttt{in}\ CrExpr(e_2))\rangle & \text{if } g = \langle e_1\ \texttt{! in}\ e_2\rangle \\
\langle CrExpr(e_1)\ \texttt{=}\ CrExpr(e_2)\rangle & \text{if } g = \langle e_1\ \texttt{=}\ e_2\rangle \\
\langle \texttt{!}\ (CrExpr(e_1)\ \texttt{=}\ CrExpr(e_2))\rangle & \text{if } g = \langle e_1\ \texttt{!=}\ e_2\rangle \\
\langle CrFmla(f_1)\ \texttt{\&\&}\ CrFmla(f_2)\rangle & \text{if } g = \langle f_1\ \texttt{\&\&}\ f_2\rangle \\
\langle CrFmla(f_1)\ \texttt{||}\ CrFmla(f_2)\rangle & \text{if } g = \langle f_1\ \texttt{||}\ f_2\rangle \\
\langle CrFmla(f_1)\ \texttt{=>}\ CrFmla(f_2)\rangle & \text{if } g = \langle f_1\ \texttt{=>}\ f_2\rangle \\
\langle CrFmla(f_1)\ \texttt{<=>}\ CrFmla(f_2)\rangle & \text{if } g = \langle f_1\ \texttt{<=>}\ f_2\rangle \\
\langle \texttt{no}\ CrExpr(e)\rangle & \text{if } g = \langle \texttt{no}\ e\rangle \\
\langle \texttt{lone}\ CrExpr(e)\rangle & \text{if } g = \langle \texttt{lone}\ e\rangle \\
\langle \texttt{one}\ CrExpr(e)\rangle & \text{if } g = \langle \texttt{one}\ e\rangle \\
\langle \texttt{some}\ CrExpr(e)\rangle & \text{if } g = \langle \texttt{some}\ e\rangle \\
\langle CrQFmla(f)\rangle & \text{if } g = \langle \texttt{all}\ \ q\ \ \ldots\rangle \\
\langle CrQFmla(f)\rangle & \text{if } g = \langle \texttt{no}\ \ q\ \ \ldots\rangle \\
\langle CrQFmla(f)\rangle & \text{if } g = \langle \texttt{lone}\ \ q\ \ \ldots\rangle \\
\langle CrQFmla(f)\rangle & \text{if } g = \langle \texttt{some}\ \ q\ \ \ldots\rangle \\
\langle CrQFmla(f)\rangle & \text{if } g = \langle \texttt{one}\ \ q\ \ \ldots\rangle \\
\langle CrExpr(e_2)\,[CrExpr(e_1)]\rangle & \text{if } g = \langle e_1\ \ \texttt{.}\ \ e_2\rangle \\
\langle CrExpr(e_2)\,[CrExpr(e_1)]\rangle & \text{if } g = \langle e_2\,[e_1]\rangle \\
\langle \{CrFmla(f_1)\ \ CrFmla(f_2)\ \ \ldots\ \ CrFmla(f_3)\}\rangle & \text{if } g = \langle \{e_1\ \ f_2\ \ \ldots\ \ f_3\}\rangle \\
\langle CrFmla(b)[\vec{n} := \vec{m}]\rangle & \text{if } g = \langle \texttt{let}\ n_1\ \texttt{=}\ m_1,\ \ldots,\ n_h\ \texttt{=}\ m_h\ \ b\rangle \\
\langle CrFmla(f)[\vec{n} := \vec{m}]\rangle & \text{if } g = \langle \texttt{let}\ n_1\ \texttt{=}\ m_1,\ \ldots,\ n_h\ \texttt{=}\ m_h\ \mid\ f\rangle
\end{cases}$$

Alloy permits use of both symbols and keywords for expressing logical formulas. Core Alloy only permits use of the symbols. In order to make the above definition more concise, the following transformations are implicitly made:

- All occurrences of **`not`** are replaced with **`!`** .
- All occurrences of **`and`** are replaced with **`&&`** .
- All occurrences of **`or`** are replaced with **||** .
- All occurrences of **`implies`** are replaced with **`=>`** .
- All occurrences of **`iff`** are replaced with **`<=>`** .

Definition 4.1.27 (Translation of Quantifier Formulas: *CrQFmla*)**.** Let f, an Alloy quantifier formula, equal one of the following two (semantically-equivalent) phrases:

- $\langle w \;\; q_1 \;\; \ldots \;\; q_h \;\; f \,\rangle$
- $\langle w \;\; q_1 \;\; \ldots \;\; q_h \;\; | \;\; f \,\rangle$

where:

- $w \in \{\texttt{all}, \texttt{no}, \texttt{lone}, \texttt{one}, \texttt{some}\}$ is the "quantification keyword".
- $\{q_1 \ldots q_h\} \subseteq \mathcal{Q}_A$ are the "quantifier variable declarations".
- For the first syntactic case: $f \in \mathcal{B}_A$ is the "body".
- For the second syntactic case: $f \in \mathcal{F}_A$ is the "body".

When $w = \texttt{all}$, $CrQFmla(f) =$

```
all  $CrDecl(q_1)$ ... $CrDecl(q_h)$ {
     { $CrQuantVarCnstr(q_1)$ ... $CrQuantVarCnstr(q_h)$ } =>
     $CrFmla(f)$
}
```

When $w \in \{\texttt{no}, \texttt{lone}, \texttt{one}, \texttt{some}\}$, $CrQFmla(f) =$

```
$q$ $CrDecl(q_1)$ ... $CrDecl(q_h)$ {
     $CrQuantVarCnstr(q_1)$ ... $CrQuantVarCnstr(q_h)$
     $CrFmla(f)$
}
```

4.1.28 Example (Translating a Quantifier Formula)**.** Let $f =$

```
all x, y : A | q[x, y]
```

Then, $CrQFmla(f) =$

```
all x : _univ, y : _univ {
    { (x in A) (y in A) } =>
    q[x, y]
}
```

Definition 4.1.29 (Extracting Constraints from Quantified Variable Declarations: *CrQuantVarCnstr*)**.** Let $d =$

$\left[\ \texttt{disj}\ \right]\ n_1\ \ldots\ n_h\ \texttt{:}\ e$

be an Alloy declaration, where:

- n_1 , … , $n_h \in \mathcal{N}$ are the "declared names".
- $e \in \mathcal{E}_A$ is the "declared type".

Then, the constraint $CrQuantVarCnstr(d)$ extracted from d is:

$\texttt{((}n_1\ \texttt{+}\ \ldots\ \texttt{+}\ n_h\texttt{)}\ \texttt{in}\ e\texttt{)}\ \texttt{\&\&}\ CrDeclDisjCnstr(d)$

Definition 4.1.30 (Translation of Expressions: *CrExpr*)**.** Given an Alloy paragraph $x \in \mathcal{E}_A$, its translation $CrExpr(x)$ is defined as:

$$
CrExpr(x) = \begin{cases}
\langle \texttt{none} \rangle & \text{if } x = \langle \texttt{none} \rangle \\
\langle \texttt{iden} \rangle & \text{if } x = \langle \texttt{iden} \rangle \\
\langle \texttt{univ} \rangle & \text{if } x = \langle \texttt{univ} \rangle \\
\langle n \rangle & \text{if } x = \langle n \rangle \\
\langle \sim CrExpr(e) \rangle & \text{if } x = \langle \sim e \rangle \\
\langle CrExpr(e_1)\ \texttt{+}\ CrExpr(e_2) \rangle & \text{if } x = \langle e_1\ \texttt{+}\ e_2 \rangle \\
\langle CrExpr(e_1)\ \texttt{-}\ CrExpr(e_2) \rangle & \text{if } x = \langle e_1\ \texttt{-}\ e_2 \rangle \\
\langle CrExpr(e_1)\ \texttt{\&}\ CrExpr(e_2) \rangle & \text{if } x = \langle e_1\ \texttt{\&}\ e_2 \rangle \\
\langle CrJoin(x) \rangle & \text{if } x = \langle e_1\,\texttt{.}\ e_2 \rangle \\
\langle CrJoin(x) \rangle & \text{if } x = \langle e_2\,\texttt{[}e_1\texttt{]} \rangle \\
\langle CrExpr(e_1)\ \texttt{->}\ CrExpr(e_2) \rangle & \text{if } x = \langle e_1\ \texttt{->}\ e_2 \rangle \\
\langle CrExpr(e_1)\ \texttt{<:}\ CrExpr(e_2) \rangle & \text{if } x = \langle e_1\ \texttt{<:}\ e_2 \rangle \\
\langle CrExpr(e_1)\ \texttt{:>}\ CrExpr(e_2) \rangle & \text{if } x = \langle e_1\ \texttt{:>}\ e_2 \rangle \\
\langle CrFmla(f)\ \texttt{=>}\ CrExpr(e_1)\ \texttt{else}\ CrExpr(e_2) \rangle & \text{if } x = \langle f\ \texttt{=>}\ e_1\ \texttt{else}\ e_2 \rangle \\
\langle CrExpr(e)[\vec{n} := \vec{m}] \rangle & \text{if } x = \langle \texttt{let}\ n_1\ \texttt{=}\ m_1\texttt{,}\ \ldots\texttt{,}\ n_h\ \texttt{=}\ m_h\ \texttt{\{}\ e\ \texttt{\}} \rangle \\
\langle CrExpr(e)[\vec{n} := \vec{m}] \rangle & \text{if } x = \langle \texttt{let}\ n_1\ \texttt{=}\ m_1\texttt{,}\ \ldots\texttt{,}\ n_h\ \texttt{=}\ m_h\ \texttt{|}\ e \rangle
\end{cases}
$$

Definition 4.1.31 (Translation of Set Comprehensions: *CrSetCompr*)**.** Let x, an Alloy set comprehension, equal one of the following two (semantically-equivalent) phrases:

- $\langle \texttt{\{}q_1\texttt{,}\ \ldots\texttt{,}\ q_h\ f\ \texttt{\}} \rangle$
- $\langle \texttt{\{}q_1\texttt{,}\ \ldots\texttt{,}\ q_h\ \texttt{|}\ f\ \texttt{\}} \rangle$

where:

- $\{q_1 \ldots q_h\} \subseteq \mathcal{Q}_A$ are the "quantifier variable declarations".
- For the first syntactic case: $f \in \mathcal{B}_A$ is the "body".
- For the second syntactic case: $f \in \mathcal{F}_A$ is the "body".

Then, the translation $CrSetCompr(x)$ of x is given by:

$$\texttt{\{}\ CrDecl(q_1)\texttt{,}\ \ldots\texttt{,}\ CrDecl(q_h)\ \texttt{\{}$$
$$CrQuantVarCnstr(q_1)\ \ldots\ CrQuantVarCnstr(q_h)\quad CrFmla(f)$$
$$\texttt{\}}\ \texttt{\}}$$

Definition 4.1.32 (Translation of Joins: *CrJoin*)**.** Let x, an Alloy join expression, equal one of the following two phrases:

- $\left\langle e_{h-1}\ \texttt{.}\ \texttt{(}e_{h-2}\ \texttt{.}\ \ldots\ \texttt{.}\ \texttt{(}e_2\ \texttt{.}\ e_1\texttt{)}\ \ldots\texttt{)}\right\rangle$
- $\left\langle e_1\texttt{[}e_2\texttt{,}\ \ldots\texttt{,}\ e_{h-1}\texttt{,}\ e_h\texttt{]}\right\rangle$

where $e_1,\ldots,e_h \in \mathcal{E}_A$ are Alloy expressions. Although these two phrases look different, they are actually semantically equivalent. That is, a series of dot joins is equal to a box join in the “reverse order”. Either phrasing can be used both for relational joins and for calls to functions and predicates.

Core Alloy is more consistent. It requires that that the first phrasing (“dot join”) is used only for relational joins, and the second phrasing (“box join”) is used only for function and predicate application.

Let us first consider the case that e_1 is an incompletely-applied function or predicate. That is, e_1 is a function or predicate reference that does not have all its parameters applied. We box join e_2 to e_1 (thereby “calling” e_1), and then invoke *CrJoin* again on the resulting expression with the rest of the arguments box joined to it. Formally, $CrJoin(x) = CrJoin\left\langle \texttt{(}e_1\texttt{[}CrExpr(e_2)\texttt{])}\texttt{[}e_3\texttt{,}\ \ldots\texttt{,}\ e_{h-1}\texttt{,}\ e_h\texttt{]}\right\rangle$.

If e_1 is any other type of expression, we simply dot join the inner expression to the translation of e_1. Formally, $CrJoin(x) = \left\langle CrJoin\left\langle e_h\ \texttt{.}\ \texttt{(}e_{h-1}\ \texttt{.}\ \ldots\ \texttt{.}\ \texttt{(}e_3\ \texttt{.}\ e_2\texttt{)}\ \ldots\texttt{)}\right\rangle\ \texttt{.}\ CrExpr(e_1)\right\rangle$.

4.1.33 Example (Translating a Series of Joins)**.** Given the following:

- `f` is the name of a function that takes 2 arguments and returns a relation with arity 3.
- `a`, `b`, `c`, `d`, and `e` are local variables with arities of 1.
- $x = \langle$`e.((c.f[a, b])[d])`$\rangle$.

Then, $\mathit{CrJoin}(x) = \langle$`e . (d . f[a][b][c])`$\rangle$.

4.2 Core Alloy to Intermediate Language

The translator to the Intermediate Language (IL) is the most mathematically interesting phase of the translation. Here the relational algebra of (Core) Alloy is translated into higher order logic, represented by the IL, a lambda calculus. While individual Core Alloy terms are translated into higher order objects, the translation of a formula is always first-order, so any IL specification that this translation could generate will be in normal form once it is normalized. The translation to IL converts Core Alloy formulas into IL formulas, and Core Alloy expressions into abstractions (i.e. anonymous functions). Declared relations from Core Alloy of type $T_1 \times \cdots \times T_n$ are represented by IL declared functions of type $(T_1 \times \cdots \times T_n) \rightarrow \mathbb{B}$.

4.2.1 Notation (Translation Functions)**.** In this section, we describe the translation to the Intermediate Language using a number of mutually-recursive functions:

- *TrSpec*: Translate from a Core Alloy specification to an IL theory.
- *TrVoc*: Translate from a Core Alloy vocabulary to a series of IL sort and function declarations.
- *TrCmd*: Translate from a Core Alloy command to a series of IL function declarations and axioms.
- *TrDirRun*: Translate from a Core Alloy directly-run predicate to a series of IL function declarations and axioms.

- $TrFmla$: Translate from a Core Alloy formula to a boolean-typed IL term.
- $TrExpr$: Translate from a Core Alloy expression to a function-typed IL
- $TrPredRef$: Translate from a Core Alloy referenced predicate to an IL term.
- $TrFuncRef$: Translate from a Core Alloy referenced function to an IL term.

Definition 4.2.2 (Translation of Specifications: $TrSpec$)**.** Let t be a Core Alloy specification consisting of v, P, U, F, and c, where:

- $v \in \mathcal{V}_C$ is the vocabulary of s.
- $P \subseteq \mathcal{P}_C$ is the set of predicate definitions in s.
- $U \subseteq \mathcal{U}_C$ is the set of function definitions in s.
- $F \subseteq \mathcal{B}_C$ is the set of formula blocks that make up the facts of s.
- $c \in \mathcal{C}_C$ is the command in s.

Recalling that an IL theory is a pair containing a set of function declarations and a set of axioms, we define the translation $TrSpec(t)$ of s as such:

$$TrSpec(t) = < D_B \cup D_V \cup D_C \ , \ A_F \cup A_C >$$

where:

- D_B is the base signature (i.e., the set of predefined functions), which is defined below.
- $D_V = TrVoc(v)$ is the set of function declarations resulting from the translation of the Core Alloy vocabulary.
- $A_F = \{TrFmla(f) \mid f \in F\}$ are the IL axioms that are the translations of the Core Alloy fact blocks.
- $< D_C, A_C >= TrCmd(c)$ are, respectively, the sets of IL function declarations and IL axioms that are the translation of the Core Alloy command.

Definition 4.2.3 (Base Signature: D_B)**.** The base signature is the set of predefined IL functions that are included in every IL signature. It contains the following, along with the shorthand we use to denote them in this section:

1. Logical Negation
    - *Formal Declaration:* $N : \mathbb{B} \to \mathbb{B}$
    - *Shorthand:* $\neg\, a$ denotes Na.
2. Logical Conjunction
    - *Formal Declarations:* $\{A_n : \mathbb{B}^n \to \mathbb{B} \mid n \geq 0\}$
    - *Shorthand:* $a \,\wedge\, b$ denotes $A_2 < a, b >$.
3. Logical Disjunction
    - *Formal Declarations:* $\{O_n : \mathbb{B}^n \to \mathbb{B} \mid n \geq 0\}$
    - *Shorthand:* $a \,\vee\, b$ denotes $O_2 < a, b >$.
4. Logical Implication
    - *Formal Declaration:* $I :< \mathbb{B}, \mathbb{B} > \to \mathbb{B}$
    - *Shorthand:* $a \;\Rightarrow\; b$ denotes $I < a, b >$.
5. Logical Equivalence
    - *Formal Declaration:* $L :< \mathbb{B}, \mathbb{B} > \to \mathbb{B}$
    - *Shorthand:* $a \;\Leftrightarrow\; b$ denotes $L < a, b >$.
6. Universal Quantification
    - *Formal Declarations:* $\{F_{n_1 \ldots n_k} : (< U^{n_1} \,,\, \ldots \,,\, U^{n_k} > \to \mathbb{B}) \to \mathbb{B} \mid n \geq 0\}$
    - *Shorthand:* $\forall\, t_1 \ldots t_k . f$ denotes $F_{Ar(t_1) \ldots Ar(t_k)}(\lambda < t_1 \,,\, \ldots \,,\, t_k > . \, f)$.
7. Existential Quantification
    - *Formal Declarations:* $\{X_{n_1 \ldots n_k} : (< U^{n_1} \,,\, \ldots \,,\, U^{n_k} > \to \mathbb{B}) \to \mathbb{B} \mid n \geq 0\}$
    - *Shorthand:* $\exists\, t_1 \ldots t_k . f$ denotes $X_{Ar(t_1) \ldots Ar(t_k)}(\lambda < t_1 \,,\, \ldots \,,\, t_k > . \, f)$.
8. Equality
    - *Formal Declarations:* $\{E_{T,n} : T^n \to \mathbb{B} \mid n \geq 0\}$
    - *Shorthand:* $s \;=\; t$ denotes $E_{T,2} < s, t >$, where T is contextually implied.
9. Inequality
    - *Formal Declarations:* $\{Q_{T,n} : T^n \to \mathbb{B} \mid n \geq 0\}$
    - *Shorthand:* $s \;\neq\; t$ denotes $Q_{T,2} < s, t >$, where T is contextually implied.

Definition 4.2.4 (Translation of Vocabularies: *TrVoc*)**.** Let $v =$

> **`sig`** `_univ {`n_1 `: _univ`^{k_1}`, ... ,` n_h `: _univ`^{k_h} `}`
>
> **`sig`** s_1 `, ... ,` s_i `{}` **`in`** `_univ`

be a Core Alloy vocabulary, where:

- $r_1 \ldots r_h \in \mathcal{N}$ are the “relation names”.
- $k_1 \ldots k_h \in \mathbb{N}$ are the arities that correspond to $r_1 \ldots r_h$.
- $s_1 \ldots s_i \in \mathcal{N}$ are the “sig names”.

Then, we define the translation $TrVoc(v)$ of v as such:

$$TrVoc(v) = R' \cup S'$$

, where:

- $R' = \{r_1 : (U^{(k_1+1))} \to \mathbb{B} \text{ , } \ldots \text{ , } r_h : (U^{(k_h+1)} \to \mathbb{B}\}$ is the set of translated relation declarations.
- $S' = \{\texttt{is-}s_1 : (U^1 \to \mathbb{B}) \text{ , } \ldots \text{ , } \texttt{is-}s_i : (U^1 \to \mathbb{B})\}$ is the set of translated sig declarations.

Definition 4.2.5 (Translation of Commands: *TrCmd*)**.** Given a Core Alloy command $c \in \mathcal{C}_C$, we define its translation $TrCmd(c)$ as such:

$$TrCmd(c) = \begin{cases} <\ \{\}\ ,\ TrFmla(b)\ > & \text{if } c = \langle \textbf{run}\ b\ \ \textbf{for}\ z\ \ \texttt{\_univ}\rangle; b \in \mathcal{B}_C, z \in \mathbb{N} \\ TrDirRun(p), \text{where } n \text{ names } p & \text{if } c = \langle \textbf{run}\ n\ \ \textbf{for}\ z\ \ \texttt{\_univ}\rangle; n \in \mathcal{N}, z \in \mathbb{N} \end{cases}$$

Definition 4.2.6 (Translation of Directly-Run Predicates: *TrDirRun*)**.** Let $p =$

> **`pred`** n`[`a_1 `: _univ`^{k_1}`, ... ,` a_h `: _univ`^{k_h}`]` b

be the definition of a directly-run predicate p. Then,

$$TrDirRun(p) =<\ D\ ,\ \{a\}\ >$$

where:

- $D = \{n@a_i : U^{k_i} \rightarrow \mathbb{B} \mid i > 0\}$ is the set of function declarations generated by the predicate parameters.
- $a = (\mathit{TrPredRef}(n)\ n@a_1\ \ldots\ n@a_h)$ is a formula. It applies the translation of the predicate reference, which is a function, to $n@a_1$, … , $n@a_h$, yielding a boolean value.

4.2.7 Example (Translating a Command that Directly Runs a Predicate)**.** Given the following:

```
pred _p_proxy[a : _univ, b : _univ -> _univ] { ... }

run _p_proxy for 8 _univ
```

The translation of the **run** command equals $< D, \{a\} >$, where:

- $D = \{$ `_p_proxy@a` : $U^1 \rightarrow \mathbb{B}$, `_p_proxy@b` : $U^2 \rightarrow \mathbb{B}$ $\}$.
- $a = (\mathit{TrPredRef}($`_p_proxy`$)$ `_p_proxy@a` `_p_proxy@b`$)$.

4.2.8 Notation (Symbols Used in Term Translation)**.** Within the definitions of TrFmla and TrExpr:

- All instances of n represent an element of $\mathcal{N}$, i.e. a name.
- All instances of s or s_h (for $h > 0$) represent an element of S, where $S \subseteq \mathcal{N}$ is the set of sig names in the Core Alloy specification.
- All instances of e or e_h (for $h > 0$) represent an element of $\mathcal{E}_C$, i.e. a Core Alloy expression.
- All instances of f or f_h (for $h > 0$) represent an element of $\mathcal{F}_C$, i.e. a Core Alloy formula.
- All instances of m or m_h (for $h > 0$) represent an element of $(\mathcal{E}_C \cup \mathcal{F}_C)$, i.e. a Core Alloy formula or expression.

Definition 4.2.9 (Translation of Formulas: $TrFmla$)**.** Given a Core Alloy formula g, we define its translation $TrFmla(g)$ as:

$$TrFmla(g) = \begin{cases}
TrPredRef(f)e_1 \ldots e_n & \text{if } g = \langle p[e_1, \ldots, e_n] \rangle \text{, where } p \text{ is a predicate} \\
TrFmla(f_1) \wedge \ldots \wedge TrFmla(f_n) & \text{if } g = \langle \texttt{\{} f_1 \ \ldots f_n \texttt{\}} \rangle \\
\neg\, TrFmla(f) & \text{if } g = \langle \textbf{\texttt{not}}\ f \rangle \\
TrFmla(f_1) \Leftrightarrow TrFmla(f_2) & \text{if } g = \langle f_1 \ \texttt{<=>}\ f_2 \rangle \\
\forall x. TrExpr(e_1)x \Rightarrow TrExpr(e_2)x & \text{if } g = \langle e_1 \ \textbf{\texttt{in}}\ e_2 \rangle \\
TrFmla(f_1) \vee TrFmla(f_2) & \text{if } g = \langle f_1 \ \textbf{\texttt{||}}\ f_2 \rangle \\
TrFmla(f_1) \wedge TrFmla(f_2) & \text{if } g = \langle f_1 \ \textbf{\texttt{\&\&}}\ f_2 \rangle \\
TrFmla(f_1) \Rightarrow TrFmla(f_2) & \text{if } g = \langle f_1 \ \texttt{=>}\ f_2 \rangle \\
\forall x. TrExpr(e_1)x \Leftrightarrow TrExpr(e_2)x & \text{if } g = \langle e_1 \ \texttt{=}\ e_2 \rangle \\
\forall x_1 \ldots x_k . TrFmla(f) & \text{if } g = \langle \textbf{\texttt{all}}\ x_1\texttt{,}\ \ldots\texttt{,}\ x_k \ \texttt{:}\ \ \texttt{\_univ} \mid f \rangle \\
\forall x_1 \ldots x_k . \neg f & \text{if } g = \langle \textbf{\texttt{no}}\ x_1\texttt{,}\ \ldots\texttt{,}\ x_k \ \texttt{:}\ \ \texttt{\_univ} \mid f \rangle \\
\forall \vec{z_1}\vec{z_2}. f[\vec{x} := \vec{z_1}] \wedge f[\vec{x} := \vec{z_2}] \Rightarrow (\vec{z_1} = \vec{z_2}) & \text{if } g = \langle \textbf{\texttt{lone}}\ x_1\texttt{,}\ \ldots\texttt{,}\ x_k \ \texttt{:}\ \ \texttt{\_univ} \mid f \rangle \\
TrFmla(\texttt{(}\textbf{\texttt{lone}}\ x_1\texttt{,}\ \ldots\texttt{,}\ x_k \ \texttt{:}\ \ \texttt{\_univ} \mid f\texttt{)} & \\
\quad \textbf{\texttt{\&\&}}\ \texttt{(}\textbf{\texttt{some}}\ x_1\texttt{,}\ \ldots\texttt{,}\ x_k \ \texttt{:}\ \ \texttt{\_univ} \mid f\texttt{)}) & \text{if } g = \langle \textbf{\texttt{one}}\ x_1\texttt{,}\ \ldots\texttt{,}\ x_k \ \texttt{:}\ \ \texttt{\_univ} \mid f \rangle \\
\exists x_1 \ldots x_n . TrFmla(f) & \text{if } g = \langle \textbf{\texttt{some}}\ x_1\texttt{,}\ \ldots\texttt{,}\ x_k \ \texttt{:}\ \ \texttt{\_univ} \mid f \rangle \\
\forall x. \neg\, TrExpr(e)x & \text{if } g = \langle \textbf{\texttt{no}}\ e \rangle \\
\exists x_1 x_2 . (TrExpr(e)x_1 \wedge TrExpr(e)x_2) \Rightarrow (x_1 = x_2) & \text{if } g = \langle \textbf{\texttt{lone}}\ e \rangle \\
TrFmla(\textbf{\texttt{lone}}\ e \textbf{\texttt{\&\&}}\ \textbf{\texttt{some}}\ e) & \text{if } g = \langle \textbf{\texttt{one}}\ e \rangle \\
\exists x. TrExpr(e)x & \text{if } g = \langle \textbf{\texttt{some}}\ e \rangle \\
TrFmla(\textbf{\texttt{\#}}\ \texttt{S <= z}) \wedge \exists x_1 \ldots x_z . & \\
\quad (\texttt{is-}S(x_1) \wedge \ldots \wedge \texttt{is-}S(x_z) \wedge Q_{S,z} < x_1, \ldots, x_z >) & \text{if } g = \langle \textbf{\texttt{\#}}\ \texttt{S = z} \rangle \text{, where } z \in \mathbb{N} \\
\forall x_1 \ldots x_{z+1} . (\texttt{is-}S(x_1) \wedge \ldots \wedge \texttt{is-}S(x_{z+1})) \Rightarrow & \\
\quad (\neg\, Q_{S,z+1} < x_1, \ldots, x_{z+1} >) & \text{if } g = \langle \textbf{\texttt{\#}}\ \texttt{S <= z} \rangle \text{, where } z \in \mathbb{N}
\end{cases}$$

Definition 4.2.10 (Translation of Expressions: $TrExpr$)**.** Given a Core Alloy expression x, we define its translation $TrExpr(x)$ as:

$$TrExpr(x) = \begin{cases} n & \text{if } x = \langle n \rangle \text{, where } n \text{ names a constant} \\ n & \text{if } x = \langle n \rangle \text{, where } n \text{ names a variable} \\ n & \text{if } x = \langle n \rangle \text{, where } n \text{ names a relation} \\ \lambda z.\ \texttt{is-}n(z) & \text{if } x = \langle n \rangle \text{, where } n \text{ names a sig} \\ TrFuncRef(n)e_1 \ldots e_n & \text{if } x = \langle n\texttt{[}e_1 \texttt{ , } \ldots \texttt{ , } e_n\texttt{]} \rangle \\ \lambda z.\ \pi^1 z \ = \ \pi^2 z & \text{if } x = \langle \textbf{\texttt{iden}} \rangle \\ \lambda z.\ true & \text{if } x = \langle \textbf{\texttt{univ}} \rangle \\ \lambda z.\ false & \text{if } x = \langle \textbf{\texttt{none}} \rangle \\ \lambda z.\ TrExpr(e_1)z \vee TrExpr(e_2)z & \text{if } x = \langle e_1 \texttt{ + } e_2 \rangle \\ \lambda z.\ TrExpr(e) < \pi^2 z, \pi^1 z > & \text{if } x = \langle \texttt{~} e \rangle \\ \lambda z.\ TrExpr(e_1)z \wedge TrExpr(e_2)z & \text{if } x = \langle e_1 \texttt{\&}\ e_2 \rangle \\ \lambda z.\ TrExpr(e_1)z \wedge \neg\, TrExpr(e_2)z & \text{if } x = \langle e_1 - e_2 \rangle \\ \lambda z.\ \exists\, w. TrExpr(e_1) < \pi^1 z, \ldots, \pi^{Ar(e_1)-1} z, w > & \\ \qquad \wedge\, TrExpr(e_1) < w, \pi^{Ar(e_1)} z, \ldots, \pi^{Ar(e_1)+Ar(e_2)-1} z > & \text{if } x = \langle e_1 \texttt{ . } e_2 \rangle \\ \lambda z.\ TrExpr(d)\pi^1 z \wedge TrExpr(r)z & \text{if } x = \langle d \texttt{<:}\ r \rangle \\ \lambda z.\ TrExpr(d)\pi^{Ar(d)} z \wedge TrExpr(r)z & \text{if } x = \langle r \texttt{:>}\ d \rangle \\ \lambda z.\ TrExpr(e_2)z & \\ \qquad \vee\, (TrExpr(e_1)z \wedge \neg\, \exists\, y. TrExpr(e_2)y \wedge \pi^1 y \ = \ \pi^1 z) & \text{if } x = \langle e_1 \texttt{++}\ e_2 \rangle \\ \lambda z.\ TrExpr(e_1) < \pi^1 z, \ldots, \pi^{Ar(e_1)} z > & \\ \qquad \wedge\, TrExpr(e_2) < \pi^{Are_1+1} z, \ldots, \pi^{Ar(e_1)+Ar(e_2)} z > & \text{if } x = \langle e_1 \texttt{->}\ e_2 \rangle \\ \lambda z.\ TrFmla(f) & \text{if } x = \langle \texttt{\{ } x \texttt{ : \_univ | } f \texttt{ \}} \rangle \end{cases}$$

Definition 4.2.11 (Translation of Predicate References: $TrPredRef$)**.** Let $p =$

$$\textbf{\texttt{pred}}\ \ n\texttt{[}a_1 \texttt{ : \_univ}^{k_1}\texttt{, ... , } a_j \texttt{ : \_univ}^{k_h}\texttt{]}\ b$$

be the definition of a predicate, where:

- $n \in \mathcal{N}$ is the name of the predicate.
- $a_1 , \dots , a_j \in \mathcal{N}$ are the "parameter names".
- $k_1 , \dots , k_j \in \mathbb{N}$ are the "parameter arities".

Then,

$$TrPredRef(n) = \lambda a_1 \dots a_j.\, TrFmla(b)$$

where, in the IL:

- $a_1 : (U^{k_1} \to \mathbb{B}) , \dots , a_j : (U^{k_j} \to \mathbb{B})$.
- $TrPredRef(n) : \mathbb{B}$

4.2.12 Example (Translating a Predicate Reference)**.** Consider the following predicate definition:

```
pred disjoint[a : _univ, b : _univ] { no (a & b) }
```

Then, $TrPredRef(\texttt{disjoint}) = \lambda ab.\, \forall\, x. \neg\, ((a\, x) \,\wedge\, (b\, x))$.

Definition 4.2.13 (Translation of Function References: *TrFuncRef*)**.** Let $f =$

fun n[a_1 : _univ^{k_1}, ... , a_k : _univ^{k_h}] : _univ$^{\ell}$ { e }

be the definition of a function, where:

- $n \in \mathcal{N}$ is the name of the function.
- $a_1 , \dots , a_j \in \mathcal{N}$ are the "parameter names".
- $k_1 , \dots , k_j \in \mathbb{N}$ are the "parameter arities".
- $\ell \in \mathbb{N}$ is the "return type arity".
- $e \in \mathcal{E}_C$ is an expression that is the "return value", where $Ar(e) = \ell$.

Then,

$$TrFuncRef(n) = \lambda a_1 \dots a_j.\, TrExpr(e)$$

where, in the IL:

- $a_1 : (U^{k_1} \to \mathbb{B}) , \dots , a_j : (U^{k_j} \to \mathbb{B})$.

- $TrFuncRef(n) : U^{\ell} \to \mathbb{B}$

4.2.14 Example (Translating a Function Reference)**.** Consider the following function definition:

```
fun union3[a : _univ, b : _univ, c : _univ] { a + b + c }
```

Then, $TrFuncRef(\texttt{union3}) = \lambda abcx.\ ((a\,x)\ \vee\ (b\,x)\ \vee\ (c\,x))$.

4.3 Normalization of the Intermediate Language

We will prove in chapter 5 that the Intermediate Language is strongly normalizing and confluent. These properties mean that *any* reduction strategy will eventually lead to a single normal form theory. We will now outline the particular normalization strategy we used.

To normalize a term, we walk through the tree the term represents, converting any redexes we find. A redex can be converted if the term being applied is an abstraction. The output of the translator to IL is designed such that when it is reduced, no abstractions will remain. This is necessary because SMT-LIB does not support anonymous functions. In fact, this is necessary because abstractions are higher order objects, and SMT-LIB is first-order. The fact that the translator to IL outputs IL that, when normalized, will have no abstractions remaining, means that the output of the translator to IL really is first order, even though it is being expressed in a higher order language.

For an IL term of the form ts, where t is of type $S \to V$ and s is of type S, the normalizer first normalizes t, naming the resulting term t', and then checks if t' is an abstraction. If it is, the normalizer beta reduces the redex and normalizes the result. If t' is not an abstraction, the normalized form of ts is simply $t's'$, where s' is s normalized.

However the Haskell representation of the IL that we used actually allows abstractions to take multiple arguments, so an application is of the form `App t [`$s_1,\ldots,s_n$`]`. If the result of `norm t` is an abstraction, the result of `norm (App t [`$s_1,\ldots,s_n$`])` is simply the result of normalizing the result of beta reduction of the arguments `[`$s_1,\ldots,s_n$`]` into `t'`. If `norm t` is not an abstraction, then `norm (App t [`$s_1,\ldots,s_n$`])` is just `App (norm t) [norm `s_1`, ..., norm `s_n`]`.

The normalization algorithm makes calls to `betaReduce`, a procedure for capture-avoiding beta reduction, which when given a term that is a redex `(lambda `$\vec{x}$` v)`$\vec{s}$ returns $v[\vec{x} := \vec{s}]$. The normalization algorithm also makes calls to the `map` procedure, whose first argument is a function which it applies to each member of its second argument, which is a list.

The definition of the normalization algorithm on terms is given by:

```
norm t =
    Case t = (Var x) :  t
    Case t = (Const x) :  t
    Case t = (Tuple ts) :  Tuple (map norm ts)
    Case t = (Lambda params body) :  Lambda params (norm body)
    Case t = (App f args) :
        let res = norm f
        If res is of the form (Lambda params body)
        then
            norm (betaReduce res args)
        else
            App res (map norm args)
    Case t = (Forall decls body) :  Forall decls (norm body)
    Case t = (Exists decls body) :  Exists decls (norm body)
    Case t = (Project t i) :  Project (norm t) i
```

One can see that the entire algorithm is a straightforward tree traversal, except for the case handling applications, which identifies redexes and reduces them.

Now we state two lemmas that, together, show that after normalization, no abstractions will remain in an IL specification generated by the translation to IL.

Lemma 4.3.1. A top-level term t generated by the normalization process is of order 0 and is in normal form, and every constant and free variable occurring in t has order 0 or 1.

Proof. Clear by inspection of the cases of the normalization algorithm, above. ///

Lemma 4.3.2. Let t be an IL term such that

1. t is of order 0;
2. every constant and free variable occurring in t has order 0 or 1;
3. Quantification is over variables of order 0.
4. t is irreducible.

Then no subterm of t is an abstraction.

Proof. The proof is by induction on t.

- If t is a constant or variable there is nothing to prove.
- We cannot have t itself be an abstraction t is of order 0.
- If t is a tuple $< t_1, \dots, t_k >$ then each t_i is of order 0, and the result holds by the induction hypothesis applied to each t_i.
- If t is an application, write t as $t_0 t_1 \dots t_n$ where $n > 0$ and t_0 is not an application, then t_0 cannot be an abstraction since t is in β-normal form. So t_0 is either a variable or a constant. By assumption, t_0 has first-order type and so $t_1, \dots, t_n$ are of order 0. So the induction hypothesis applies to each t_i and the result follows.

- If t is a projection $\pi^i t'$ then t' cannot be a tuple since t is irreducible. So t' is of the form $t_0 t_1 \ldots t_n$ where $n > 0$ and t_0 is not an application. We can apply the same argument as in the application case above.

- If t is an existential quantification $\exists x_1 \ldots x_n.b$ then (since we only quantify over type-0 variables) the induction hypothesis applies to b and the result follows. The same reasoning applies when t is a universal quantification.

///

With `norm` defined, the definition of the normalization of IL theories becomes trivial.

Definition 4.3.3 (Translation of IL Theories: *NrmTheory*)**.** Let $t =< v, A >$ be an IL theory.

The normalization $NrmTheory(t)$ of t equals $< v, \{\texttt{norm}\, a \mid a \in A\} >$.

4.4 Normalized Intermediate Language to SMT

Once the Intermediate Language specification is normalized, it can be reduced to SMT-LIB. This stage of the pipeline performs four operations:

1. Check that there are no abstractions remaining, and raise an error if there are.

2. Reduce tuples to multiple singletons.

3. Rename symbols that would be syntactically invalid in SMT-LIB.

4. Move from IL data structures to SMT-LIB data structures.

Of these, only the second operation is mathematically interesting. Converting tuples to singletons is done by converting the tuple of terms to a list of terms. Where terms accept a list of arguments, we simply flatten the list of lists given by translating its list of arguments (e.g., [[1], [2, 3]] becomes [1, 2, 3]). Where terms must have an argument that is only one term, we simply unpack the list and see that it has exactly one element, throwing an error otherwise.

The equals and not equals operations represent a special challenge. For two tuples to be equal, they must be of the same length, and each element of the first tuple must be equal to the corresponding element of the second tuple. That is, for $a =< a_1, \ldots, a_n >$ and $b =< b_1, \ldots, b_n >$, $a = b$ if and only if $a_1 = b_1 \wedge \cdots \wedge a_n = b_n$. Comparing two tuples for inequality, i.e. $a \neq b$, is similar: $a \neq b$ if and only if $a_1 \neq b_1 \vee \cdots \vee a_n \neq b_n$.

The pseudocode we will give below refers to the following helper functions:

- **`concat:`** given a list of lists, flattens it into a single list. For example, `concat [[1], [2, 3], [4, 5]] = [1, 2, 3, 4, 5]`.
- **`ttsVarRef:`** given an IL variable reference that is to a tuple, returns a list containing a variable reference for each element of the tuple. For any other variable reference, simply returns a list containing the variable reference.
- **`ttsVarDecl:`** given an IL variable declaration for a tuple, returns a list containing a variable declaration for each element of the tuple. For any other variable declaration, simply returns a list containing the declaration.
- **`ttsConstRef:`** given a constant, determines if it's a built-in. If so, applies the appropriate special behavior. Otherwise, translates constant references the same way variable references are translated.

The definition of the algorithm for translation of normalized IL terms to SMT-LIB is given in cases by:

```
ttsmt t =
   Case t = (Var x) :  ttsVarRef t
   Case t = (Const x) :  ttsConstRef t
   Case t = (Tuple ts) :  concat (map ttsmt ts)
   Case t = (Lambda params body) :  error
   Case t = (App f args) :
      let args' = concat (map ttsmt args)
      App f args'
   Case t = (Forall decls body) :
      let decls' = concat (map ttsVarDecl decls)
      let body' = ttsmt body
      If length body' == 1
      then
         let body' = body'[1]
      else
         error
      Forall decls' body'
   Case t = (Exists decls body) :  ((Identical to Forall))
   Case t = (Project t i) :  (ttsterm t)[i]
```

We then go on to define $TsTheory$, which describes how to translate from IL theories to SMT theories, along with two helper functions $TsDecl$ and $TsZeroOrderType$.

Definition 4.4.1 (Translation of IL Theories: $TsTheory$)**.** Let $t =< v,A >$ be a normalized IL theory.

The translation $TsTheory(t)$ of t is an SMT theory with:

- The sorts U and $\mathbb{B}$.
- The function declarations $\{TsDecl(d) \mid d \in v\}$.
- The axioms $\{\texttt{ttsmt}\ a \mid a \in A\}$.

Definition 4.4.2 (Translation of IL Function Declarations: $TsDecl$)**.** Consider an normalized IL function declaration d, which takes the form $n : T \to V$.

The translation $TsDecl(d)$ of d is an SMT declaration with the name n and the type $TsZeroOrderType(T) \to TsZeroOrderType(V)$.

Definition 4.4.3 (Translation of Zero-Order IL Types: $TsZeroOrderType$)**.** Let t be an IL type.

If $t = \mathbb{B}$ or $t = U$, then $TsZeroOrderType(t) = t$.

Otherwise, if $t = T_1 \times \cdots \times T_n$, then:

- Let $V_{1,1} \times \cdots \times V_{1,m_1} = TsZeroOrderType(T_1)$.
- $\ldots$
- Let $V_{n,1} \times \cdots \times V_{n,m_n} = TsZeroOrderType(T_n)$.

Then, $TsZeroOrderType(t) = V_{1,1} \times \cdots \times V_{1,m_1} \times \cdots \times V_{n,1} \times \cdots \times V_{n,m_n}$.

Chapter 5

Properties of the Intermediate Language

The Intermediate Language is a simply-typed lambda calculus. It is defined in section 3.3.

5.1 Definitions

The proofs in this section will rely on the following definitions.

First, to make the arguments in this section simpler and cleaner, we will work with a variation on the Intermediate Language whose terms are as follows:

Definition 5.1.1 (Terms of the Constant-Based IL)**.** An IL term will take one of the following forms:

1. **Constant:** A constant term is an ordered pair $< n,t >$ of a name n and a type t. Declared functions are constants. A constant can be of any type.

2. **Variable:** A variable term is an ordered pair $< n,t >$ of a name n and a type t. A variable can be of any type.

3. **Tuple:** Given terms $t_1, \ldots, t_k$ ($k \geq 0$) whose types are $T_1, \ldots, T_k$, the term $< t_1, \ldots, t_k >$ is a tuple of type $< T_1, \ldots, T_k >$. The terms of a tuple do not need to be of the same type.

4. **Abstraction:** Given a variable x whose type is T, and a term v whose type is V, $\lambda x.v$ is a function of type $T \rightarrow V$.

5. **Application:** Given a term t of type $S \rightarrow V$ and a term s of type S, ts is a term of type V.

6. **Projection:** Given a term $< t_1, \ldots, t_k >$ of type $< T_1, \ldots, T_k >$, and an integer i, $\pi^i t$ is a term of type T_i.

We call this language the "constant-based IL," as opposed to the IL used in the translation itself, which can be thought of as the "constructor-based IL." In section 5.2, where we prove that this constant-based IL is strongly normalizing, we will also show that if this IL is strongly normalizing, than the constructor-based IL used by the translation must be also.

Definition 5.1.2 (Reduction Relation)**.** Given any set T and a binary relation $\rightarrow$ on T, $\rightarrow$ is a *reduction relation*.

Definition 5.1.3 (Strong Normalization)**.** A reduction relation $\rightarrow$ is *strongly normalizing* if every sequence of terms $t \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots$ is finite.

Definition 5.1.4 (Confluence)**.** A reduction relation $\rightarrow$ is *confluent* if for all terms t, t_1, t_2 such that $t \rightarrow t_1$ and $\rightarrow t_2$, there exists a term w such that $t_1 \rightarrow w$ and $t_2 \rightarrow w$.

Definition 5.1.5 (ν)**.** If t is a strongly normalizing term, consider the tree representing all possible reductions from t, where each node is a term reachable from t and each edge connects a term a to a term b such that $a \rightarrow b$. Then, $\nu(t)$ is the number of edges in this tree.

Note that for a term t that is normal, $\nu(t) = 0$.

Definition 5.1.6. A term t is ***neutral*** if it is

- c, a constant
- x, a variable
- $\pi^i t$, a projection
- ts, an application

Note that this is equivalent to saying that t is *not* of the form $\lambda x.v$ or $< t_1, \dots, t_n >$.

Definition 5.1.7 (R_T)**.** R_T is the set of terms that are 'reducible of type T.' A term t of type T is a member of R_T if

1. If T is atomic (U or $\mathbb{B}$), $t \in R_T$ if t is strongly normalizing.
2. If T is a tuple type $< T_1, \dots, T_n >$, $t \in R_T$ if $\pi^i t \in R_{T_i}$ $\forall i \in \{1, \dots, n\}$
3. If $T = S \to V$, an arrow type, $t \in R_T \Leftrightarrow \forall s \in R_S, ts \in R_V$

5.2 Proof of Strong Normalization

This section is a proof that the Intermediate Language is strongly normalizing (SN). That is, for every term t, every sequence of reductions starting from t terminates.

The fact that the lambda calculus is strongly normalizing was first proven by Tait, who showed the proof in [Tai67]. We show an adaptation of the proof to the Intermediate Language. The form of the proof is heavily inspired by its presentation by Girard in chapter 6 of [GLT89].

First we show that proving SN for our constant-based version of the IL is sufficient. To do this we first give the following definition and prove the following lemma:

Definition 5.2.1 (f)**.** f is a map from constructor-based IL terms to constant-based IL terms. It sends $\forall x.b$ to $\forall\lambda x.f(b)$, and similarly sends $\exists x.b$ to $\exists\lambda x.b$, and leaves other terms unchanged.

Lemma 5.2.2. Given a reduction sequence $t_0 \to t_1 \to \dots$ of constructor-based IL terms, this map induces a reduction sequence $t'_0 \to t'_1 \to \dots$ of constant-based IL terms.

Proof. It is clear that from the sequence $\{t_i\}$ the map induces a parallel sequence $\{f(t_i)\}$, but we must show that for any i, $t_i \to t_{i+1}$ implies that $f(t_i) \to f(t_{i+1})$.

This is clear for constructor-based IL terms besides $\forall x.b$ and $\exists x.b$, since the map f does not change those other terms.

For terms of the form $\forall x.b$ and $\exists x.b$, which map to terms of the form $\forall\lambda x.f(b)$ and $\exists\lambda x.f(b)$, respectively, all reduction sequences occur entirely within the term b. Here there are two cases:

1. Neither b nor any subterm of b is $\forall$ or $\exists$. Then reduction sequences starting from b and reduction sequences starting from $f(b)$ are identical.

2. Otherwise, we can see inductively (with (1) as the base case) that for reduction sequences starting from b, their analogues starting from $f(b)$ will still be present.

Thus $t_i \to t_{i+1}$ does imply that $f(t_i) \to f(t_{i+1})$.

///

Note that we do not need to (and do not) preclude the possible existence of two constructor-based IL terms t_1 and t_2 such that t_1 does *not* reduce to t_2, but $f(t_1) \to f(t_2)$.

With the above map and definition in place, we can prove

Theorem 5.2.3. If the constant-based Intermediate Language, defined in section 5.1 above, is strongly normalizing, then the constructor-based Intermediate Language, defined in section 3.3, is also strongly normalizing.

Proof. By lemma 5.2.2 reduction sequences in the constructor-based IL map to reduction sequences of the same length or longer in the constant-based IL. But by assumption, reduction sequences in the constant-based IL are always finite, so therefore reduction sequences in the constructor-based IL must also always be finite. ///

Now we can seek to prove that the constant-based IL, simply called "the IL" hereafter, is Strongly Normalizing (SN). There are three lemmas that will be needed to complete the proof. First,

Lemma 5.2.4. The following are true:

(F1) $t \in R_T \Leftrightarrow$ t is strongly normalizing.

(F2) $t \in R_T$ and $t \twoheadrightarrow t' \Rightarrow t' \in R_T$

(F3) If t is neutral and all reductions of a redex in t yield a term $t' \in R_T$, then $t \in R_T$

Note that a special case of (F3) is that if t is neutral and normal, then the second condition of (F3) is vacuously true and so $t \in R_T$.

Proof. By simultaneous induction on the types of terms.

Base Case: Let t a term of atomic type T.

(F1) By definition of R_T

(F2) Notice that t must be strongly normalizing. Any term t' st $t \twoheadrightarrow t'$ must be also.

(F3) Assume conversion of any redex in t results in a term t' such that $t' \in R_T$. Let a reduction path r be given. It must pass through one of the t'. But these are strongly normalizing, so r must be of finite length. In fact, $(\max_{t' \, s.t. \, t \to t'} \nu(t')) + 1 = \nu(t)$.

Since t is strongly normalizing, it is in R_T by definition.

Inductive Step: There are two remaining cases: T is either a tuple type, or an arrow type.

Tuple Type: t = $< t_1, \ldots, t_n >$

(F1) $t \in R_T \Rightarrow t_i \in R_{T_i} \forall i$ by definition of R_T. By the induction hypothesis, the t_i are strongly normalizing, and $\nu(t_i)$ is defined $\forall i$.

Note $\nu(t) \leq \nu(\pi^i t) \forall i$ since for any reduction sequence $t, t', \ldots$ the parallel sequence $\pi^i t, \pi^i t', \ldots$ where the projection is not reduced also exists. But t_i strongly normalizing implies $\pi^i t$ strongly normalizing $\forall i$, i.e. $\nu(\pi^i t)$ exists and therefore $\nu(t) \leq \nu(\pi^i t) \forall i \Rightarrow \nu(t)$ exists, i.e. $t \in R_T$.

(F2) $t \twoheadrightarrow t' \Rightarrow \pi^i t \twoheadrightarrow \pi^i t \; \forall i.$ t is reducible by hypothesis and therefore so is $\pi^i t \; \forall i.$ Thus by (F2) of the induction hypothesis, $\pi^i t'$ is reducible $\forall i$, which is the definition of $t' \in R_T$.

(F3) Take $T = < T_1, \ldots, T_n >$. Let t neutral and all t' such that $t \rightarrow t'$ are in R_T. Note that t neutral implies t is not a tuple term. Therefore, any single step reduction of $\pi^i t$ will yield a term of the form $\pi^i t'$. $t' \in R_T \Rightarrow \pi^i t' \in R_{T_i}$. $\pi^i t$ is neutral, so since all one step reductions yield a reducible term, we have by the inductive hypothesis that $\pi^i t \in R_{T_i} \; \forall i$, but that is the definition of $t \in R_T$.

Arrow Type:

(F1) Take $T = S \rightarrow V$. Given $t \in R_{S \rightarrow V}$, let x of type S. By the inductive hypothesis (F3), x neutral and normal $\Rightarrow x \in R_S$. Thus $tx \in R_V$ by definition. By the inductive hypothesis, tv is strongly normalizing, i.e. $\nu(tv)$ is finite.

Note that for every reduction sequence $t \rightarrow t' \rightarrow \ldots$ there is a parallel reduction sequence $tv \rightarrow t'v \rightarrow \ldots$ in which the redex of t applied to v is not reduced. Therefore $\nu(t) \leq \nu(tv)$. But since $\nu(tv)$ is finite this means that $\nu(t)$ is also, i.e. t is strongly normalizing.

(F2) Take $T = S \to V$. Let $t \in R_{S\to V}$ and t' such that $t \twoheadrightarrow t'$. Want $t' \in R_{S\to V}$.

Let $x \in R_S$. $t \in R_{S\to V} \Leftrightarrow tx \in R_V$, so $tx \in R_V$. Note that $tx \twoheadrightarrow t'x$. By the induction hypothesis (F2), $t'x \in R_V$, but $(t'x \in R_V \forall x \in R_S) \Leftrightarrow t' \in R_{S\to V}$

(F3) Take $T = S \to V$. Let t neutral and all t' such that $t \to t'$ are in R_T. Let $x \in R_S$. Want $tx \in R_V$

By (F1) of the induction hypothesis, x is strongly normalizing, i.e. $\nu(x)$ is finite. Note that t neutral means that t is *not* of the form $\lambda a.b$. Therefore tu converts in one reduction step to either $t'x$ or tx'.

<u>t' x:</u> We have $t' \in R_T$, and $x \in R_S$ also. By definition, $t' \in R_T \Leftrightarrow (\forall x, x \in R_S \Rightarrow t'x \in R_V)$.

<u>t x':</u> By (F2) of the inductive hypothesis, $x' \in R_S$. Note that $\nu(x') < \nu x$. By the inductive hypothesis for x', tx' is reducible.

We can see that in either case tx converts to reducible terms only. (F3) for type V then shows that tx is reducible. But x was arbitrary, so t is reducible by definition.

///

Lemma 5.2.5 (Pairing). If $t_i \in R_{T_i}\ \forall i$, then $< t_1, \dots, t_n > \in R_{<T_1,\dots,T_n>}$

Proof. Note that by (F1), $\nu(t_i)$ exists for every i. Proceed by induction on $\sum_{i=1}^{n} \nu(t_i)$:

Base Case: $\sum_{i=1}^{n} \nu(t_i) = 0$

Since $\nu(t)$ can never be negative, we have that $\nu(t_i) = 0\ \forall i$. Therefore t_i is reducible and normal. By (F3), $\pi^i < t_1, \dots, t_n >$ is reducible, and so $< t_1, \dots, t_n >$ is reducible by definition.

Inductive Step:

$\pi^i < t_1, \dots, t_n >$ converts in one reduction step to

- t_i, assumed to be reducible.

- $\pi^i < t_1, \ldots, t'_j, \ldots, t_n >$, with $t_j \to t'_j$. We supposed that t_j is reducible, so by (F2) t'_j is reducible. $\nu(t'_j) < \nu(t_j)$, so $< t_1, \ldots, t'_j, \ldots, t_n >$ is reducible by the inductive hypothesis.

$\forall i, \pi^i < t_1, \ldots, t_n >$ is neutral and all one step reductions lead to a reducible term, so by (F3) it is reducible. Therefore $< t_1, \ldots, t_n >$ is reducible by definition. ///

Lemma 5.2.6 (Abstraction)**.** If $\forall s \in R_S$, $v[x := s] \in R_V$, then $(\lambda x.v) \in R_{S \to V}$.

Proof. By definition, $(\lambda x.v) \in R_{S \to V} \Leftrightarrow (\forall s \in R_S, (\lambda x.v)s \in R_V)$

Let $s \in R_S$ be given. We can then proceed by induction on $\nu(s) + \nu(v)$:

Base Case: $\nu(s) + \nu(v) = 0$ Since $\nu(x) \geq 0\ \forall x$, we must have $\nu(v) = 0$ and $\nu(s) = 0$, i.e. v and s are normal. $(\lambda x.v)s$ is neutral and normal so by (F3) it is in R_V. Since s was arbitrary, we have $\forall s$ such that $\nu(s) = 0, s \in R_S \Rightarrow ts \in R_V \Leftrightarrow t \in R_{S \to V}$.

Inductive Step:

$(\lambda x.v)s$ converts to one of the following:

$v[x := s]$: reducible by hypothesis.

$(\lambda x.v')s$: for $v \to v'$. (F2) implies $v' \in R_V$, so (F1) implies that $\nu(v')$ exists. Clearly $\nu(v') < \nu(v)$, so by the inductive hypothesis $(\lambda x.v')s \in R_V$.

$(\lambda x.v)s'$: for $s \to s'$. (F2) implies $s' \in R_S$. So by (F1), $\nu(s')$ exists. $\nu(s') < \nu(s)$, so by the inductive hypothesis this is reducible.

Every term t' such that $(\lambda x.v)s \to t'$ is reducible, so by (F3) it is reducible. But s was arbitrary, so we have that $\forall s \in R_S, (\lambda x.v)s \in R_V \Leftrightarrow \lambda x.v \in R_{S \to V}$. ///

With these in place, we are ready to prove that all terms are strongly normalizing. We will do this by proving that all terms are reducible (which implies by (F1) that all terms are strongly normalizing). However to do this we need a slightly stronger inductive hypothesis, so we first must prove the following:

Lemma 5.2.7. Let t be any term (not necessarily reducible), and name all the free variables in t $\vec{x} = x_1, \ldots, x_n$, whose types are $T_1, \ldots, T_n$, respectively. If $\vec{s} = s_1, \ldots, s_n$ are reducible terms of types $T_1, \ldots, T_n$, then $t[\vec{x} := \vec{s}]$ is reducible.

Proof. By induction on t. There are 5 cases:

1. t is c, a constant. A constant is clearly normal, and it is neutral, so by (F3), it is reducible.

2. t is x_i, a var. So $t[\vec{x} := \vec{s}] = s_i$, which is reducible.

3. t is $\pi^i t'$, a projection of a term t' of tuple type. By the inductive hypothesis, $t'[\vec{x} := \vec{s}] \in R_T$, and therefore so is $\pi^i t'[\vec{x} := \vec{s}] = t[\vec{x} := \vec{s}]$.

4. t is $< t_1, \ldots, t_n >$. By the inductive hypothesis, $t_1, \ldots t_n$ are reducible, so by lemma 5.2.5 t is also.

5. t is wv. By the inductive hypothesis $w[\vec{x} := \vec{s}]$ and $v[\vec{x} := \vec{s}]$ are reducible. So $(w[\vec{x} := \vec{s}])v[\vec{x} := \vec{s}] \in R_T$ by the definition of reducibility, but that term is just $t[\vec{x} := \vec{s}]$.

6. t is $\lambda y.w$ of type $T = V \to W$. By the inductive hypothesis, we have $\forall v : V, w[\vec{x} := \vec{s}, y := v] \in R_W$. This satisfies the hypothesis of lemma 5.2.6, which states that $t[\vec{x} := \vec{s}] \in R_T$.

///

Note that all variables are reducible. Therefore one can substitute the free variables for themselves to find that any term is reducible. So by (F1), any term is strongly normalizing.

5.3 Proof of Confluence

We will prove in this section that the Intermediate Language, which is a simply typed lambda calculus, is confluent. While we will not show it here, it happens to be true that a lambda calculus does not need to be strongly normalizing, or simply typed, to be confluent.

We seek to prove that the Intermediate Language is *confluent*, i.e. that for all terms t, t_1, t_2 such that $t \twoheadrightarrow t_1$ and $t \twoheadrightarrow t_2$, there exists a term w such that $t_1 \twoheadrightarrow w$ and $t_2 \twoheadrightarrow w$. However, there is a similar claim, called *weak confluence*:

Definition 5.3.1 (weak confluence)**.** for all terms t, t_1, t_2 such that $t \rightarrow t_1$ and $t \rightarrow t_2$, there exists a term w such that $t_1 \twoheadrightarrow w$ and $t_2 \twoheadrightarrow w$.

Note that these two definitions are not equivalent. Take for example the lambda calculus with four unique terms a, b, c, and d, with the reduction relation $a \rightarrow b$, $b \rightarrow a$, $a \rightarrow c$, and $b \rightarrow d$. This calculus is weakly confluent but not confluent.

However, weak confluence and confluence *are* equivalent in the presence of strong normalization (the example reduction relation above is clearly *not* strongly normalizing; there is a reduction path starting with $a \rightarrow b \rightarrow a \rightarrow \dots$ that does not terminate). This result is called Newman's Lemma, and its first proof was given by Newman in [New42]. We repeat the proof here:

Theorem 5.3.2. Let $\rightarrow$ be some reduction relation. If

1. $\rightarrow$ is strongly normalizing, and

2. $\forall t, t_1, t_2$ such that $t \rightarrow t_1$ and $t \rightarrow t_2$, $\exists w$ such that $t_1 \twoheadrightarrow w$ and $t_2 \twoheadrightarrow w$,

then $\forall t, t_1, t_2$ such that $t \twoheadrightarrow t_1$ and $t \twoheadrightarrow t_2$, $\exists w$ such that $t_1 \twoheadrightarrow w$ and $t_2 \twoheadrightarrow w$.

Proof. Let t, s, v such that $t \twoheadrightarrow s$ and $t \twoheadrightarrow v$ be given. There is a reduction sequence $t \to t_1 \to \cdots \to s$ and a reduction sequence $t \to t_2 \to \cdots \to v$. Since $\to$ is weakly confluent by hypothesis, there is a term w such that $t_1 \twoheadrightarrow w$ and $t_2 \twoheadrightarrow w$.

Proceed by induction on $\nu(t)$:

Since $\nu(t_1) < \nu(t)$ and both s and w reduce from t_1, we have by the inductive hypothesis that there is some term u_1 such that $s \to u_1$ and $w \to u_1$.

Since $\nu(t_2) < \nu(t)$ and both u_1 (through w) and v reduce from t_2, we have by the inductive hypothesis that there is some term u such that $u_1 \twoheadrightarrow u$ and $v \twoheadrightarrow u$.

But $s \to u_1 \to u$ also, i.e. u is such that $s \to u$ and $v \to u$. ///

So we can set out to prove that the Intermediate Language is weakly confluent, i.e.:

Theorem 5.3.3. For any term t, for all t_1, t_2 such that $t \to t_1$ and $t \to t_2$, there exists a term w such that $t_1 \twoheadrightarrow w$ and $t_2 \twoheadrightarrow w$.

To do this, we will rely on the following three lemmas:

Lemma 5.3.4. If s_1, s_2, and t are terms, x is a free variable in that might appear in s_2 and t, and y is a free variable that might appear in t, then $t[y := s_2][x := s_1] = t[x := s_1][y := s_2[x := s_1]]$.

Proof. By induction on t.

- t is a constant. x and y do not appear in t, so $t[y := s_2][x := s_1] = t$ and $t[x := s_1][y := s_2[x := s_1]] = t$.

- t is a variable. If it is neither x nor y, the situation is just as the constant case.

  Otherwise, if $t = x$, then $t[y := s_2][x := s_1] = s_1$ and $t[x := s_1][y := s_2[x := s_1]] = s_1[y := s_2[x := s_1]]$, but y does not appear in s_1 by hypothesis so $s_1[y := s_2[x := s_1]] = s_1$.

If $t = y$, then $t[y := s_2][x := s_1] = t[y := s_2[x := s_1]] = s_2[x := s_1]$ and $t[x := s_1][y := s_2[x := s_1]] = s_2[x := s_1]$.

- t is a tuple $< t_1, \ldots, t_n >$. $t[y := s_2][x := s_1] \ = \ < t_1, \ldots, t_n > [y := s_2][x := s_1] \ = \ < t_1[y := s_2][x := s_1], \ldots t_n[y := s_2][x := s_1] >$.

  By the inductive hypothesis, $< t_1[y := s_2][x := s_1], \ldots t_n[y := s_2][x := s_1] > = \ < t_1[x := s_1][y := s_2[x := s_1]] t_n[x := s_1][y := s_2[x := s_1]] >$, but that is just $t[x := s_1][y := s_2[x := s_1]]$.

- t is a projection $\pi^i t'$. By the inductive hypothesis, $t'[y := s_2][x := s_1] = t'[x := s_1][y := s_2[x := s_1]]$, and clearly $\pi^i b[x := s] = \pi^i b[x := s]$ for any s, b, and i, so $t[y := s_2][x := s_1] = t[x := s_1][y := s_2[x := s_1]]$.

- t is an abstraction $\lambda x.b$. By the inductive hypothesis, $b[y := s_2][x := s_1] = b[x := s_1][y := s_2[x := s_1]]$, and clearly $\lambda x.b[x := s] \ = \ \lambda x.b[x := s]$ for any s, b, and i, so $t[y := s_2][x := s_1] = t[x := s_1][y := s_2[x := s_1]]$.

- t is a redex $(\lambda x.b)s$. $((\lambda x.b)s)[y := s_2][x := s_1]$ is just $(\lambda x.b[y := s_2][x := s_1])(s[y := s_2][x := s_1])$.

  But by the inductive hypothesis, $b[y := s_2][x := s_1] = b[x := s_1][y := s_2[x := s_1]]$ and $s[y := s_2][x := s_1] = s[x := s_1][y := s_2[x := s_1]]$, so $t[y := s_2][x := s_1] = t[x := s_1][y := s_2[x := s_1]]$.

///

Lemma 5.3.5. Let s, s' such that $s \to s'$ be given. Let b be any term. Let x be some variable of the same type as s that is free in b. Then $b[x := s] \twoheadrightarrow b[x := s']$

Proof. By induction on b.

- b is a constant. So x does not appear in b, so s does not appear in $b[x := s]$. Thus $b[x := s] = b[x := s']$, which of course means that $b[x := s] \twoheadrightarrow b[x := s']$.

- b is a variable. If $b \neq x$, it is just as in the constant case. Otherwise, $b[x := s] = s$. But $s \to s'$ by hypothesis, so $b[x := s] \to b[x := s']$.

- b is a tuple $< b_1, \ldots, b_n >$. By the inductive hypothesis, $b_i[x := s] \twoheadrightarrow b_i[x := s']$ for all i, so $< b_1, \ldots, b_n > [x := s] \twoheadrightarrow < b_1, \ldots, b_n > [x := s']$.

- b is a projection $\pi^i b'$. By the inductive hypothesis, $b'[x := s] \twoheadrightarrow b'[x := s']$. Then, $\pi^i b'[x := s] = \pi^i b[x := s] \to \pi^i b[x := s'] = \pi^i b[x := s']$.

- b is an abstraction $\lambda y.v$. By the inductive hypothesis, $v[x := s] \twoheadrightarrow v[x := s']$, but $\lambda y.v[x := s] = \lambda y.v[x := s] \to \lambda y.v[x := s'] = \lambda y.v[x := s']$.

- b is a redex $(\lambda y.v)r$. By the inductive hypothesis, $v[x := s] \twoheadrightarrow v[x := s']$ and $r[x := s] \twoheadrightarrow r[x := s']$, but $((\lambda y.v)r)[x := s] = (\lambda y.v[x := s])(r[x := s]) \to (\lambda y.v[x := s'])(r[x := s']) = ((\lambda y.v)r)[x := s']$.

///

Lemma 5.3.6. Let b, b' be given such that $b \to b'$. Let x be some variable that is free in b. Finally let s be a term of the same type as x. Then, $b[x := s] \twoheadrightarrow b'[x := s]$

Proof. By induction on b.

- b is a constant or a variable: This cannot happen. Because constants and variables are always normal, no such b' could exist.

- b is a tuple $< b_1, \ldots, b_n >$: so b' is of the form $< b_1, \ldots, b_i', \ldots, b_n >$, for b_i' such that $b_i \to b_i'$. By the inductive hypothesis, $b_i[x := s] \to b_i'[x := s]$, so $b[x := s] \twoheadrightarrow b'[x := s]$.

- b is a projection $\pi^i v$: So there are two cases.

- v is a tuple $< v_1, \ldots, v_n >$ and the reduction is converting b to v_i. Clearly $v[s := x] \to v_i[x := s]$, so $b[x := s] \twoheadrightarrow b'[x := s]$.
- Otherwise, b' is of the form $\pi^i v'$ for some v' such that $v \to v'$. By the inductive hypothesis, $v[x := s] \to v'[x := s]$, so $b[x := s] \twoheadrightarrow b'[x := s]$.

- b is an abstraction $\lambda y.v$: So b' is of the form $\lambda y.v'$ for v' such that $v \to v'$. By the inductive hypothesis, $v[x := s] \to v'[x := s]$, so $b[x := s] \twoheadrightarrow b'[x := s]$.

- b is an application $l\ r$. There are two cases:
  - l is an abstraction $\lambda y.v$, and b' is of the form $v[y := r]$. Recall that substitutions are written such that the *outermost* substitution is the one that should be applied first. We have that $(lr)[x := s] \to v[y := r[x := s]][x := s]$, and by lemma 5.3.4, $v[y := r][x := s] = v[x := s][y := r[x := s]]$, so $(l\ r)[x := s] \to (v[y := r])[x := s]$.
  - Otherwise, we can simply use the inductive hypothesis. $l[x := s] \to l'[x := s]$ and $r[x := s] \to r'[x := s]$ for r' and l' such that $r \to r'$ and $l \to l'$, so $(lr)[x := s] = b[x := s] \to b'[x := s]$.

///

Now we are ready to prove theorem 5.3.3, i.e. for any term t, for all t_1, t_2 such that $t \to t_1$ and $t \to t_2$, there exists a term w such that $t_1 \twoheadrightarrow w$ and $t_2 \twoheadrightarrow w$.

Proof. Let such terms t, t_1, t_2 be given. Proceed by induction on t. There are five cases:

- t is a variable or a constant. Such a term is normal so there can be no such t_1 and t_2.

- $t =< v_1, \ldots, v_n >$, a tuple. So t_1 is of the form $< v_1, \ldots, v_i', \ldots, v_n >$ and t_2 is of the form $< v_1, \ldots, v_j', \ldots, v_n >$, where $v_i \to v_i'$ and $v_j \to v_j'$. There are two cases:

  - $i = j$. v_i is confluent by the inductive hypothesis.
  - $i \neq j$. Let $w =< v_1, \ldots, v'_i, \ldots, v'_j, \ldots, v_n >$. $t_1 \to w$ and $t_2 \to w$.

- $t = \pi^i t'$. Here t' is confluent by the inductive hypothesis.

- $t = \lambda x.b$. Here b is confluent by the inductive hypothesis.

- $t = (\lambda x.b)s$. There are three cases:
  - $t_1 = (\lambda x.b')s$ and $t_2 = (\lambda x.b)s'$, where $b \to b'$ and $s \to s'$. Let $w = (\lambda x.b')s'$. Clearly $t_1 \to w$ and $t_2 \to w$.
  - $t_1 = (\lambda x.b')s$ and $t_2 = b[x := s]$. Let $w = b'[x := s]$. Clearly $t_1 \to w$. And by lemma 5.3.6, $t_2 \twoheadrightarrow w$.
  - $t_1 = (\lambda x.b)s'$ and $t_2 = b[x := s]$. Let $w = b[x := s']$. Clearly $t_1 \to w$. And by Lemma 5.3.5, $t_2 \twoheadrightarrow w$.

///

Chapter 6

Code Discussion

In section 2.2, we explain the structure of the translator program at a high level. In this section, we state which tools we used to implement our design, and then reflect on the benefits and drawbacks of the design, our execution of it, and what we learned from the process.

6.1 Tools and Methodology

Git

Version control is vital for the success of a modern software project. Our team was most familiar with Git so we chose to use that version control system. We did not need the particular advantages that its competitors have, like locking files from Subversion, which was not needed for a team of two, and the general user friendliness of Mercurial, which was not needed for programmers who already knew Git.

Haskell

We wrote the translator in Haskell[1], a purely functional language with a strong type system. Haskell initially entered our consideration for the intensely practical reason that

[1] https://www.haskell.org/

our advisor and one of us (Kyle McCormick) were already experienced with Haskell. However, Haskell's merits are what led us to choose it as the primary language of the translator.

We saw three main benefits of choosing Haskell over any alternative. First, Haskell's type system is very strong, meaning that many programming mistakes are actually caught by the compiler. This is more pleasant for the programmer and also considerably increases development speed. Second, Haskell's functional and compositional nature, and its monad data type, make Haskell excellent for flexibly and quickly specifying pipelines. We knew early on that the translator would be a series of steps at the top level — lexing, then parsing, then reducing alloy to a simpler core, etc. — and it quickly became clear that the core structure of Haskell is well suited for creating pipelines. Third and finally, the monadic error reporting system makes tracking errors though a pipelines far easier: all the cruft of throwing and handling exceptions is done implicitly and consistently by the Haskell type system, accelerating development time both upfront and later on during debugging.

Stack

Haskell comes with the Cabal package manager to manage third party Haskell packages, but it has problems with encapsulation and requires a significant amount of manual input. Stack[2] is a tool built on top of Cabal that gives each project its own environment and automatically handles package dependencies. Around halfway through this project, we switched to from vanilla Cabal to using Stack to organize builds and manage dependencies. This simplified our build process and preemptively prevented issues that could have been caused by modifying packages for other projects.

Java

The Alloy Analyzer is written in Java[3]. Most of the functionality of the Alloy Ana-

[2]https://docs.haskellstack.org/en/stable/README/
[3]https://www.oracle.com/java/index.html

lyzer is available as libraries in the Alloy.jar file that holds the Analyzer. We used Java 1.8 during this project to write a simple CLI that would allow some of our Haskell programs to interact with the Alloy Analyzer.

Alex and Happy

The first stage of the pipeline is built using the Alex[4] lexer generator. Alex is a tool that takes a specification on how to split a string into tokens as input, and produces a Haskell function that perform said tokenization. Similarly, the second stage of the pipeline is built using the Happy[5] parser generator. Happy is a tool takes a specification on how to build an abstract syntax tree from a token stream as input, and produces a Haskell function that performs said parsing.

Both these tools were chosen because they are free and open source, seemed easy to use, and target Haskell as their output language.

6.2 Takeaways and Lessons Learned

Forrest had never used Haskell before this project. Learning the language was useful in and of itself, but also illustrated deep ideas about functional programming and type systems. Understanding monads will likely be very useful in the future.

We struggled with establishing clear and general naming conventions for variables and entities in this project. We often had to choose between conciseness and descriptiveness. The result of this is that our code is more difficult to read than it needed to be. Spending more time planning naming conventions more thoroughly and asking for more guidance would have been beneficial.

Testing could have been done better for this project. We made small Alloy files and wrote their expected output, but we never automated these integration tests or made unit

[4]https://www.haskell.org/alex/
[5]https://www.haskell.org/happy/

tests for smaller pieces of the code. There were several situations in which automated tests and/or unit tests obviously would have helped us. Further, better testing would have helped us be more confident to make claims about the translator's behavior.

Related to insufficient testing, we let our progression of the project fall into the big bang integration methodology: we got every piece of the pipeline sort of working, and then tied it all together at once and fixed dozens of bugs throughout the entire pipeline. It certainly would have been more efficient to clearly specify what each stage of the pipeline should do, and finish each stage of the pipeline individually, testing them one at a time before integrating the entire translator. An alternative approach would have been taking narrow vertical slices: first making the translator able to correctly translate a very small subset of Alloy, and progressively adding on more features, one at a time. We achieved this workflow sometimes, but not consistently, which slowed our progress.

Chapter 7

Evaluation of the Translator

7.1 Feature Support

In this section we outline the features of Alloy our translation does support but that our translator does not support (i.e., whose translation we defined mathematically, but whose translation we did not implement in our proof-of-concept translator). These were mainly abandoned due to time constraints.

- Defining relations whose types reference relations within *the same* sig declaration. For example, the declaration of `s` referencing `r` as its type makes the following code snippet fail in our translator: `sig A {} sig B { r: A, s: r }`. This was a software engineering issue, and was not fixed solely due to time constraints.
- Gracefully handling multiple "runs" or "checks"; currently they are just conjoined. This problem is just as much deciding what the behavior of the translator *should* be as actually implementing it. This would be another vital feature for taking the translator from an experiment and making it into a robust tool that professionals would use to analyze their Alloy specifications with SMT solvers.

- Optionally disabling implicit scope of 3 on all runs. One of the main advantages of SMT over Alloy is that SMT solvers often don't need boundedness to analyze a specification, while Alloy does. The translator would be massively more useful and informative if we could modify the translator to optionally omit the bounds. The translator could easily, if tediously, be modified to pass boolean flags and other configuration data through the entire pipeline, which would then make it trivial to conjoin certain formulas to the specification only when a given flag is set to 'false.'

7.2 Correctness Propositions

Due to lack of time, the correctness of the translation was not formally proven. This section gives a precise statement of our correctness claim, and identifies the intermediate results that comprise the proof.

7.2.1 Terminology

Recall that:

- $\mathcal{T}_A$ is the set of Alloy specifications.
- $\mathcal{T}_C$ is the set of Core Alloy specifications.
- $\mathcal{T}_I$ is the set of IL theories.
- $\mathcal{T}_S$ is the set of SMT theories.
- *CrSpec* translates from an Alloy specification to a Core Alloy specification.
- $TrSpec$ translates from a Core Alloy specification to an IL theory.
- $TsTheory$ translates from an IL theory to an SMT theory.
- Alloy / Core Alloy *specifications* and IL/SMT *theories* correspond to logical theories.
- Alloy / Core Alloy *instances* and IL/SMT *structures* are the respective interpretations of specifications and theories.

Furthermore, in order to articulate the correctness propositions, we define several terms:

Definition 7.2.1 (Elements)**.** In Alloy, Core Alloy, the IL, and SMT, *elements* are unary, scalar values.

Definition 7.2.2 (Tuples of Elements)**.** Given a set of elements E, the *tuples of E* are all the tuples $e_1 \times \cdots \times e_n$ that can be formed from all $\{e_1 \ldots e_n\} \subseteq E$.

We denote the tuples of E by $T(E)$.

Definition 7.2.3 (Alloy Instances: $\mathcal{I}_A$)**.** An Alloy instance consists of:

- A set of non-$\mathbb{B}$ elements U.
- A set of sigs, each of which contains a subset of U.
- A set of relations on U.

Definition 7.2.4 (Core Alloy Instances: $\mathcal{I}_C$)**.** A Core Alloy instance is an Alloy instance for which there exists an explicit sig, `_univ`, which includes all elements in U.

Definition 7.2.5 (IL Structures: $\mathcal{I}_I$)**.** An IL Structure M^1 is a first-order "slice" of a standard IL model M, constructed as follows:

The sorts of M^1 are the first-order sorts of M, and the M^1-interpretations of the functions are as in M. (Recall that by definition, functions declared in IL have first-order types.)

Definition 7.2.6 (SMT Structures: $\mathcal{I}_I$)**.** An SMT structure consists of:

- A set of sorts, each of which contains a set of elements.
- A set of functions from $T(E)$ to E, where E is the union of the sets of elements that make up the sorts.

7.2.2 Model Conversion

Definition 7.2.7 (Conversion of Alloy Instances: *CrInst*)**.** Let i be an Alloy instance with elements U.

The Core Alloy conversion $CrInst(i)$ of i is equal to i, with the exception of the addition of the `_univ` sig, which contains every element in U.

The following definition describes the key bridge between the semantics of Alloy and the semantics of SMT-Lib.

Definition 7.2.8 (Conversion of Core Alloy Instances: *TrInstance*)**.** Let v be a Core Alloy vocabulary, S be the names of sigs declared in v, and R be the relations declared in v.

Let i be a Core Alloy instance with the elements U.

The IL conversion $TrInstance(i)$ of the Core Alloy instance i consists of:

- The set of elements $\mathbb{U}$, which equals U.
- The following functions:
  - For each sig name $s \in S$, we have the function `is-`$s : \mathbb{U} \to \mathbb{B}$, interpreted in the obvious way.
  - For each relation name $r \in R$, which represents a k-artiy Core Alloy relation, we have the IL function $r : \mathbb{U}^{(k)} \to \mathbb{B}$, interpreted in the obvious way.

Definition 7.2.9 (Conversion of IL Structures: *TsStruct*)**.** The SMT conversion $TsStruct(s)$ of an IL structure s containing the functions F consists of:

- The sort $\mathbb{B}$, which contains the elements $true$ and $false$.
- The sort $\mathbb{U}$.
- The set of functions G, which is built by taking the functions in F and flattening their input tuples.

7.2.3 Proof of Correctness: Outline

Alloy to Core Alloy

The following proposition states that for every Alloy specification t_A, the set of *conversions of the instances that model t_A* equals the set of *instances that model the translation of t_A*.

Proposition 7.2.10 (Correctness of Translation to Core Alloy)**.** $\forall t_A \in \mathfrak{T}_A, \{CrInst(i_A) | \, i_A \in \mathfrak{I}_A \wedge i_A \models t_A\} = \{i_C | \, i_C \in \mathfrak{I}_C \wedge i_C \models CrSpec(t_A)\}$.

Proof. Since the translation to Core Alloy consists of syntax manipulations with the Alloy language, it is straightforward to verify that each transformation preserves meaning. ///

Core Alloy to Intermediate Language

The one aspect of our translation that is neither syntactic de-sugaring (as is the case for translation to Core Alloy) nor well-known transformations (such as β-reduction or tuple-flattening) is the translation from Core Alloy to the IL. It is here where the two languages have different notions of "model." Definition 7.2.8 described the bridge between these two notions. The following proposition expresses the claim that our translation is correct relative to this conversion.

Proposition 7.2.11 (Correctness of Translation to IL)**.** $\forall t_C \in \mathfrak{T}_C, \{TrInstance(i_C) | \, i_C \in \mathfrak{I}_C \wedge i_C \models t_C\} = \{i_I | \, i_I \in \mathfrak{I}_I \wedge i_I \models TrSpec(t_C)\}$.

Intermediate Language to SMT

The following proposition states normalizing an IL theory does not change its models.

Proposition 7.2.12 (Correctness of Normalization of the IL)**.** $\forall t_I \in \mathfrak{T}_I, i_I \in \mathfrak{I}_I, (t_I \models i_I) \Leftrightarrow (NrmTheory(t_I) \models i_I))$

Proof. This is simply the correctness of β-reduction. ///

The following proposition states that for every normalized IL theory t_I, the set of *translations of the structures that model* t_I equals the set of *structures that model the translation of* t_I.

Proposition 7.2.13 (Correctness of Translation to SMT)**.** $\forall t_I \in \mathcal{T}_{I'}, \{TsStruct(i_I) | \ i_I \in \mathcal{I}_I \wedge i_I \models t_I\} = \{i_S | \ i_S \in \mathcal{I}_S \wedge i_S \models TsTheory(t_I)\}$.

where $\mathcal{T}_{I'}$ indicates the set of normalized IL theories.

Proof. This is simply the correctness of the process of flattening nested tuples. ///

7.3 Testing

7.3.1 Method

Files with Multiple Commands

Some of the Alloy test files we used have multiple commands (**`run`** or **`check`**). In the Alloy Analyzer, the user chooses individual commands to test one at a time. The translator does not elegantly handle multiple commands currently, so we broke each Alloy file with *n* commands into *n* Alloy files numbered 1 to *n*, with one command each, all having the full original theory. For example, the Alloy specification

```
sig S{}
run {no S}
check {no S}
```

would become the two files

```
sig S{}
run {no S}
```

and

```
sig S{}
check {no S}
```

Tools

We used the Unix `time` command to time certain Unix commands. We used the Unix `wc` command with the `-w` argument to count the number of words in the input Alloy files and their translations. We used Java to run AlloyCLI, a Java program we made for this testing. To evaluate the SMT2 files written by the translator, we used z3, a popular open source SMT solver provided by Microsoft Research.

Testing Procedure

For each Alloy file (which now has one command) with the name `<filename>`, we ran the following shell commands:

```
time java -classpath .:alloy4.2.jar AlloyCLI <fileName>
time cat <fileName> | stack exec alloy2smt -- > out.smt2
wc -w <fileName>
wc -w out.smt2
time z3 out.smt2
```

In the above commands `alloy2smt` is the the name of the executable translator.

We recorded the time given by the three time commands (we summed the time spent in user space and the time spent in the kernel), the output given by AlloyCLI (stating what the result of the Alloy command in the given file was), the result of the two word count commands, and the result of z3 on the translated file (the translated files contain the command `(check-sat)`, which commands z3 to search for a model of the formula stated in the given file). We ran the time commands fifteen times, and present the average of the times in the table below. All tests were run on an Intel Core i3-4005U CPU at 1.70GHz.

7.3.2 Data

We collected the Alloy files we used to test the translator from Daniel Jackson's Software Abstractions ([Jac12a]). The files are also available from the book's website. We omitted the Alloy files that our translator does not support. Unfortunately transitive closure, which the translator does not support, is a widely used feature of Alloy, forcing us to omit many of the Alloy specifications from Software Abstractions.

Below is a table showing the results of testing the supported Alloy files. The second column, "Alloy Analyzer Result," will have different values depending on whether the command of that file was a **`run`** or a **`check`**. Both of these commands take a formula as an argument. When the Alloy Analyzer is given a **`run`** command, it searches for an instance (recall that what the SMT community calls a "model," the Alloy community calls an "instance," and what the SMT community calls a "specification," the Alloy community calls a "model") which satisfies the specification and the given formula. If the Analyzer finds such an instance, the result is "instance found," and otherwise it reports "no instance found." Our translation is equisatisfiable with an Alloy file with a **`run`** command if it is "sat," i.e. it has a model, when there is an Alloy instance satisfying the **`run`** command (i.e. the Analyzer reports "instance found"). If there is no instance that satisfies the theory and the **`run`** command, so Alloy Analyzer reports "no instance found," the translation would be correct if it were "unsat."

When the Alloy Analyzer is given a **`check`** command, it searches for an Alloy instance which satisfies the theory but which makes the given formula false. If the Analyzer does not find such an instance, the result is "no counterexample" and otherwise (if it can find such an instance) it reports "counterexample found." An Alloy file with a **`check`** command is equisatisfiable with its translation if the translation is "sat" when the Alloy file is "no counterexample found," and the translation is "unsat" when the Alloy file is "counterexample found."

Finally, the Alloy file barbers.als is an inconsistent specification (it is from a homework problem in [Jac12a] in which students modify the specification in different ways to make it consistent). We inserted the command **`run`** `{}` into this file, and because the Alloy specification itself is contradictory, that command returned “no instance found.” In this case as in other cases of “no instance found,” it would be correct for the translation to be “unsat.”

The table is visually separated into three sections based on which type of command the Alloy files have: first **`run`**, then **`check`**, and lastly barbers which is an inconsistent specification.

The timing commands were run fifteen times, and the entries in the table represent an average of those fifteen times.

Alloy Specification Name	**Alloy Analyzer Result**	**Alloy Analyzer Average Runtime (seconds)**	**Translator Average Runtime**	**Alloy File Number of Words**	**SMT-LIB File Number of Words**	**Translation Result**	**Translation Average Runtime (seconds)**
properties	no instance found	1.230	0.278	76	246	unsat	0.024
barbers	no instance found	0.991	0.208	31	195	unsat	0.031
AddressBook 1 †	instance found	1.305	0.207	264	393	sat	0.033
AddressBook 2	instance found	1.313	0.224	264	496	sat	0.036
AddressBook 3 †	no counterexample	1.398	0.217	264	441	sat	0.038
AddressBook 4	no counterexample	2.585	0.223	264	791	sat	0.035
AddressBook 5 †	no counterexample	1.399	0.228	264	418	sat	0.035
AddressBook 6 †	no counterexample	1.300	0.218	264	392	sat	0.028
lights 1	no counterexample	1.358	0.249	164	1085	sat	0.042
lights 2	counterexample found	1.307	0.241	164	753	unsat	0.036
lists	no counterexample	1.192	0.223	91	525	sat	0.036
sets2	no counterexample	4.152	0.216	73	735	-	timed out

† : Time entries for specifications marked with † represent one run, not an average of fifteen runs. We chose to test some of the Address Book files less thoroughly because each Address Book file is exactly identical except for the command, so we took one check command and one run command to be representative of the overall performance of the translator and its output on that specification.

7.3.3 Discussion

The translator was correct for every Alloy specification tested, i.e. the original Alloy specification and its SMT-LIB translation were equisatisfiable. The translated specifications' performance is very good: in fact, it was faster than the Alloy Analyzer every time! State of the art SMT solvers like z3 are highly optimized, so this is not a huge surprise, but it does indicate that our translation doesn't introduce any major inefficiencies. This makes it clear that a viable option for analyzing large Alloy files that the Alloy Analyzer takes prohibitively long to examine would be to first translate them into SMT and then use an SMT solver.

The translator does, however, greatly increase the length of the files, with the output having between three and ten times as many words as the original input. The Alloy Language is certainly more expressive than SMT-LIB, so it's likely some of this size increase is unavoidable, but it is also worth noting that the translator does not optimize the generated code at all, so the complexity of the output could definitely be reduced. Reducing this complexity may also increase the speed of the SMT solver even further.

Not listed in the table is the readability of the translator's output for each file, because "readability" is subjective. However in the authors' opinion the generated SMT-LIB code is extremely difficult for a human to read, and the translator could be significantly more useful and user-friendly if the output was made more readable. We elaborate on this sentiment in section 9.1, Future Work.

Chapter 8

Related Work

The use of alternative tools, especially SMT solvers, to analyze relational algebra in general and the Alloy language in particular, is an active area of research. An early attempt was made by [EGT11] who translated a subset of the Alloy language into the SMT-LIB language. The later works [EG15] and [EGTH15] from the same authors went on to address many limitations of the original work, including describing a first-order axiomatization of transitive closure that does not require finitization, which while it is naturally incomplete, works in a huge majority of cases.

Later, [MRTB17] also developed a translation that can turn Alloy specifications into code for their SMT solver CVC4. Their translation works by reducing Alloy to a relational logic that they defined and made their SMT solver able to interpret. As the authors mentioned, the advantage of this approach is that translating between two relational languages is relatively easy, so it's straightforward to translate additional languages this way.

Others used mathematically similar approaches to translate Alloy into various logics so that automated theorem provers can be used to analyze statements about Alloy specifications. For example, [Gei11] and [UGEGT12] describe a translation from Alloy to a first-order logic used by the KeY theorem prover, which can execute fully automatic and user-guided proofs.

Lastly, a major step in using theorem provers to augment or replace the Alloy Analyzer was made by [NMMB14], which describes the integration of Alloy into the Heterogeneous Tool Set (HeTS), which is an "open source, general framework for formal methods integration and proof management" [MMCL13], allowing specifications in the Alloy language to be analyzed by all the theorem provers that have also been integrated with HeTS.

Chapter 9

Conclusion and Future Work

9.1 Future Work

Significant work remains to be done to turn the proof-of-concept translation program we provide into a complete tool that professionals could use to leverage SMT solvers to analyze their Alloy specifications.

Many features of Alloy are currently unsupported, and would need to be added. The most important of these, like transitive closure and integers, are necessary for the translator to be practically useful to the formal methods community. Some of these missing features are mathematically straightforward and only require programming effort to complete. Others, like transitive closure, will be more interesting. See section 7.1 for a complete list of the features of Alloy that the translator does not yet support.

Additionally, the translator's SMT-LIB output is relatively difficult for a human to read. Things like optimization of the generated SMT-LIB code, explanatory comments, comments that relate the generated SMT-LIB code to portions of the input Alloy file, and more informative variable names would make it significantly easier for a user to gain useful information from the translation beyond just whether their theory has models, and to utilize the SMT solver more efficiently. The translator could also be improved by efforts to refactor and simplify the currently existing Haskell code: as is common in software development, many possible improvements to our approach became apparent after we had executed it.

Finally, the reliability of the translator needs to be continually verified by a robust testing suite. Model finding often serves critical verification roles, so guaranteed correctness is absolutely imperative for the translator to be useful to real users. By creating a testing harness which allows many unit and integration tests to be automatically run for every new version of the translator, we could verify and demonstrate that the translator is working as expected.

Testing of our translator showed some suggestive performance results: while the Alloy Analyzer always took over a second to analyze its specifications, Z3 was always able to analyze the translation of the specifications in 30 to 40 milliseconds. Our relatively small sample size, both in terms of distinct Alloy specifications and the number of runs we did on each one, means that these results are not definitive, but they are suggestive. If work towards integrating Alloy and SMT continues, investigation of the potential performance gains would be worthwhile.

9.2 Summary and Conclusion

In this report, we gave a mathematical description of a translation from the Alloy Language to the SMT-LIB language. This was made interesting by the fact that Alloy uses idioms from relational algebra while SMT-LIB is based on classical first order logic. We also built a software application that stands as a proof-of-concept for the translation. We then evaluated the translation program and the performance of the SMT-LIB code the translator generates. There are many avenues for future work in this area.

Bibliography

[BFT17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[BSST09] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[C$^+$] David R Cok et al. The smt-libv2 language and tools: A tutorial.

[EG15] Aboubakr Achraf El Ghazi. Relational reasoning-constraint solving, deduction, and program verification. 2015.

[EGT11] Aboubakr Achraf El Ghazi and Mana Taghdiri. Relational reasoning via smt solving. In *International Symposium on Formal Methods*, pages 133–148. Springer, 2011.

[EGTH15] Aboubakr Achraf El Ghazi, Mana Taghdiri, and Mihai Herda. First-order transitive closure axiomatization via iterative invariant injections. In *NASA Formal Methods Symposium*, pages 143–157, 2015.

[Gei11] Ulrich Geilmann. *Verifying Alloy Models Using KeY*. PhD thesis, Diplomarbeit, Karlsruhe Institute of Technology, 2011.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7. Cambridge University Press Cambridge, 1989.

[Jac12a] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[Jac12b] Daniel Jackson. *Software Abstractions: logic, language, and analysis. Appendix B: Alloy Language Reference*. MIT press, 2012. available online: http://alloy.mit.edu/alloy/documentation/book-chapters/alloy-language-reference.pdf.

[MMCL13] Till Mossakowski, Christian Maeder, Mihai Codescu, and Dominik Lucke. Hets user guide-version 0.99. *DKFI GmbH, Bremen, Germany*, 2013.

[MRTB17] Baoluo Meng, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. Relational constraint solving in smt. In *International Conference on Automated Deduction*, pages 148–165. Springer, 2017.

[New42] Maxwell Herman Alexander Newman. On theories with a combinatorial definition of “equivalence”. *Annals of mathematics*, pages 223–243, 1942.

[NMMB14] Renato Neves, Alexandre Madeira, Manuel Martins, and Luís Barbosa. An institution for Alloy and its translation to second-order logic. In *Integration of Reusable Systems*, pages 45–75. Springer, 2014.

[SLM$^+$92] Bart Selman, Hector J Levesque, David G Mitchell, et al. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, volume 92, pages 440–446, 1992.

[Tai67] William W Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.

[Tra50] Boris A Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *Doklady Akademii Nauk SSSR*, 70:569–572, 1950.

[UGEGT12] Mattias Ulbrich, Ulrich Geilmann, Aboubakr Achraf El Ghazi, and Mana Taghdiri. A proof assistant for Alloy specifications. In *TACAS: International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 422–436. Springer, 2012.

Appendices

Appendix A

Notation

Symbol	**Rule**	**Description**
$\mathcal{T}_A$	$spec$	Specifications
$\mathcal{S}_A$	$sigDecl$	Signature declarations
$\mathcal{X}_A$	$extClause?$	Signature extension clauses
$\mathcal{P}_A$	$predDef$	Predicate definitions
$\mathcal{U}_A$	$funDef$	Function definitions
$\mathcal{A}_A$	$fact$	Facts
$\mathcal{R}_A$	$assert$	Assertions
$\mathcal{C}_A$	cmd	Commands
$\mathcal{K}_A$	$scope?$	Command scopes
$\mathcal{Y}_A$	$typescope$	Command type-scopes
$\mathcal{M}_A$	$term$	Terms
$\mathcal{F}_A$	$fmla$	Formulas
$\mathcal{E}_A$	$expr$	Expressions
$\mathcal{D}_A$	$decl$	Parameter/Relation Declarations
$\mathcal{Q}_A$	$quantDecl$	Quantifier Variable Declarations
$\mathcal{N}$	$name$	Names
$\mathbb{N}$	$natNum$	Natural (≥ 0) Numbers

Table A.1: **Elements of Supported Alloy**. This table relates symbols to the parts of Alloy that they represent and to the grammar rules that define them. The grammar of Supported Alloy can be found in Definition 3.1.1.

Symbol	**Rule**	**Description**
$\mathcal{T}_C$	$spec$	Specifications
$\mathcal{V}_C$	$vocabs$	Vocabularies
$\mathcal{S}_C$	$sigDecl$	Signature declarations
$\mathcal{P}_C$	$predDef$	Predicate definitions
$\mathcal{U}_C$	$funDef$	Function definitions
$\mathcal{A}_C$	$fact$	Facts
$\mathcal{C}_C$	cmd	Commands
$\mathcal{F}_C$	$fmla$	Formulas
$\mathcal{E}_C$	$expr$	Expressions
$\mathcal{D}_C$	$decl$	Parameter/Relation Declarations
$2\ \mathcal{N}$	$name$	Names
$\mathbb{N}$	$natNum$	Natural (≥ 0) Numbers

Table A.2: **Elements of Core Alloy**. This table relates symbols to the parts of Core Alloy that they represent and to the grammar rules that define them. The grammar of Core Alloy can be found in Definition 3.2.1.