Using Scala For Computer-Assisted Stepwise Refinement of Software (from Specification To Implementation)

by Stephen D. Motty

A thesis submitted to the

School of Graduate Studies

in partial fulfilment of the

requirements for the degree of

Master of Engineering (Computer Engineering)

Faculty of Engineering and Applied Science

October 2014

St. John's Newfoundland

Abstract

It is possible to generate compilable source code directly from logical formulas that describe the intended behaviour of software. Theories in support of this goal, including theories of predicative programming and programming by specification, were developed and well-understood by the mid-1990's. In practice these techniques result in code and specification libraries that are maintainable, sharable and unmistakably fit for their purpose.

At present, the techniques have not met with widespread adoption. An underlying premise of this thesis is that adoption requires a proficient use of techniques outside the normal curriculum followed by many computer programmers. A Structured Imperative Modular Programming/proof-Language and Environment, nicknamed SIMPPLE, is proposed to pull these techniques together into a single framework.

The specific objective of the thesis is to describe how the typed trees representing a SIMPPLE document are converted to queries to third-party provers. Using the queries, these provers verify the logical correctness of the SIMPPLE program as it is constructed line by line, a process known as stepwise refinement. Two classes of provers are considered: first, Satisfiability Modulo Theorem (SMT) provers; and secondly high-order Theorem Proving Systems (TPS).

The thesis concludes that stepwise refinement of software is practiceable, and that the implementation is readily achievable with today's technology. Examples of programs created using the techniques are provided.

Acknowledgements

Thanks are extended to Dr. Theodore S. Norvell who introduced me to the many concepts developed throughout this thesis and who patiently waited for me to become a believer. The financial contributions of the Memorial University of Newfoundland (MUN) Faculty Association and the Natural Sciences and Engineering Resource Council of Canada (NSERC) are gratefully acknowledged.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

HOL	Higher-order Logic
JVM	Java Virtual Machine
MCS	Model Checking Software
MUN	Memorial University of Newfoundland
NSERC	Natural Sciences and Engineering Research Council of Canada
SIMPPLE	Structured Imperative Modular Programming/proof – Language and Environment
SMT	Satisfiability Modulo Theorem
TPS	Theorem Proving System

Chapter 0

Introduction

0.0 Motivation

A software implementation is a syntactically correct document that can be compiled into machine executable code. Arguably, the software implementation is expected to be correct in ways other than just syntax: specifically, the user's expectations of what the software will do should be fulfilled by executing the resulting code. Unfortunately, while syntactic correctness of an implementation is a prerequisite to producing executable code, the latter is not.

A behavioural specification is a mathematical expression that describes how the output of software should respond given certain guarantees about its inputs.[Gries and Schneider, 1993] Several techniques for deriving software implementations directly from specifications exist in the literature, with distinct methods proposed by Morgan [Morgan, 1994], Goguen [Goguen *et al.*, 2000] and Hehner [Hehner, 1993]. Software derived in such a manner has the advantage that it is both syntactically and

behaviourally correct.

0.1 Contributions

This thesis describes the development of a structured imperative modular programming/proof language and environment (nicknamed SIMPPLE) capable of verifying stepwise refinements of programs. The main contribution of the thesis is a Scala language implementation that interfaces to third-party provers. These provers verify the logical correctness of the SIMPPLE program as it is constructed line by line. Two classes of provers are considered: first, Satisfiability Modulo Theorem (SMT) solvers; and secondly higher-order Theorem Proving Systems (TPS). A demonstration of the capabilities of the software is provided. The demonstration consists of:

- parsing of textual documents conforming to a BNF grammar into abstract syntax trees (ASTs);
- conversion of source abstract syntax trees (ASTs) into first-order logic;
- conversion of first-order logic into target ASTs of an SMT solver; and
- an implementation plan for targeting a higher-order TPS for solution of equally or more challenging proofs.

0.2 Other Approaches

This thesis falls under the category of automated verification of software. The approach used here emphasizes the derivation of a correct program from a specification

using Hehner's method of stepwise refinement.[Hehner, 2014] Other approaches to automated verification of software include runtime verification, property specification languages, static code analysis, and model-checking.

Runtime verification is the least exhaustive of the methods and tests only the input and output relationships covered by the system's test vectors. The approach used in this thesis aims to prove the software correct for all possible initial and final states.

Property specification languages allow a programmer to annotate code line-by-line with details about what should and should not hold true as the program executes. The assertions are considered auxiliary to the program and can often be superfluous. The approach used in this thesis is to define the input and output requirements (the specification) and use logical theory to prove an equivalence between the specification and an implementation. Theorems are introduced instead of assertions; these theorems provide meaning and definitions for terms used in the specification and are not auxiliary statements. The aim is to prove the implementation sound in the presence of the theory by using its theorems where necessary to refine the specification.

Static analysis techniques detect common programming errors (e.g. buffer overflows, memory leaks, redundant loop, and branch conditions), often by graphing program flow and propagating a set of values through the graph until a fix-point is reached meaning all relevant scenarios have been considered.[d'Silva *et al.*, 2008] Static verification is helpful in determining if a program maintains legal values throughout its execution. It is more limited than the approach used in this thesis, which ensures the values are both legal and correct according to a logical formulation of the user's expectations.

"Model-checking algorithms exhaustively examine the reachable states of a program." [d'Silva *et al.*, 2008], but suffer from problems with state-space explosion where the complexity of the model grows exponentially in response to small changes in the program. "To use model-checking on any program with more than six variables requires abstraction, and each abstraction requires proof that it preserves the properties of interest." [Hehner, 2014] The techniques used in this thesis provide the means for such proofs.

0.3 Organization of Chapters

The thesis begins with this overview, the aim of which is to present the motivation behind the thesis and inform the reader as to what to expect in the coming chapters. Readers will find the thesis informative on a number of topics. Readers new to the concept of programming by specification will need to understand the background material in chapters 1 to 2. Readers interested in the mechanics of applying Scala to the problem of programming by specification will also find chapters 3 to 5 of interest. Those who already have an understanding of the nature of the problem addressed by the thesis will find chapters 6 to 8 of greater interest, as these chapters deal with modern methods of solving these problems.

The title of the thesis is "Using Scala for Computer-assisted Stepwise Refinement of Software (from Specification to Implementation)". Chapter 1 introduces each of the concepts mentioned in this title, namely:

- computer-assisted activities and the manner in which instructions are formulated in preparation for computer processing, concentrating on grammars and

syntax trees;

- Scala, which is a modern example of a programming language that incorporates features useful for translation between languages;
- step-wise methods for arriving at an implementation of a specification; and
- the production of software from specifications to implementations, including the definition of refinement.

In Chapter 2, the discussion narrows in on the specific abstract syntax used by the server software to record and represent documents processed by the automated system. This abstract syntax allows the developer to write specifications and implementations interchangeably within the context of one document.

In Chapters 3 and 4, a concrete grammar is presented along with a case study showing how a typical document is parsed. This concrete grammar is required in order to present readable examples, and to develop test cases for the software developed under this thesis. A mock-up of the client-side editor is used in Chapter 4 to demonstrate how the software might appear to a typical user.

Chapter 5 begins the discussion of the software developed under this thesis, focussing on the task of accepting the trees parsed from the concrete grammar and converting them to a normalized, first-order equivalent.

In Chapter 6, the interface by which the normalized form is prepared for processing by third-party software is described. In Chapter 7, the process of passing test cases to a specific SMT solver are presented along with results. Chapter 8 provides a plan for an alternative implementation that targets a higher-order TPS.

The thesis concludes with a review of the case studies, focussing on the immediate applications to incorporate programming loops, and additional capabilites available from the back-end oracles that can be investigated in the immediate future.

0.4 Guide to Notation

The thesis is very notation heavy and makes extensive use of notation from extended BNF grammars, Scala Programming Language, higher-order logic (e.g. Kananaskis HOL) and denotational semantics.

Extended BNF notation is used most extensively in Chapter 3 and Chapter 5. A brief introduction to extended BNF grammar is given in Section 1.1.0. Additional details regarding BNF notation is available from the Encyclopedia of Computer Science, maintained online by the Association of Computing Machinery (ACM) at `http://dl.acm.org/ralston.cfm`.

The syntax of the Scala Programming Language is described in detail at `http://www.scala-lang.org`. Relevant features of the language, along with sample source code is presented in this paper in Section 1.2. The Scala Programming syntax is used throughout the document implementation details, including the implementation of constructors for terminals described by the BNF grammars. UML diagrams such as Figures 2.4, 2.5 and 2.6 use Scala-style functional prototypes, and for consistency the sections describing these figures do also.

Examples of the use of notation from higher-order logic and denotational semantics abound in Tables 1.0. and 8.0. In Chapter 8, corresponding symbols from the theorem proving system Kananaskis HOL are used in source code. Symbols for log-

ical and ("$\wedge$") and logical or ("$\vee$") are often attributed to Russel and Heyting, and are approximated in many computer languages, including Kananaskis HOL using the compound symbols "/\" and "\/". The symbol "$\Rightarrow$" is used for logical implication, approximated in Kananaskis HOL using the compound symbol "==>". In Kananaskis HOL, the exclamation mark "!" is used in place of the symbol "$\forall$" (universal generalization, or forall); the question mark "?" is used in place of the symbol "$\exists$" (existential specification, or exists). Additionally, as introduced in the online documentation for Kananaskis HOL at `http://hol.sourceforge.net`, a lambda abstraction is used to convert an expression to a function by binding typed variables to the scope of the expression which follows. The symbol "\" is used for lambda abstractions in Kananaskis HOL instead of the more conventional symbol "λ" used elsewhere in the thesis.

Denotational semantics are used in Tables 1.0. and 8.0 and throughout Chapters 5 and 8. An example of this notation is given below:

$$\|F \Rightarrow G\| = (\|F\| \Rightarrow \|G\|)\,.$$

The vertical bars indicate that the delimited expression x is transformed from its syntax into a semanticly equivalent interpretation. We can express this in words by "the interpretation of x", or less formally, "what is meant by x". So, in the example above, the expression can be stated: the interpretation of "$F \Rightarrow G$" is 'what is meant by "F" implies what is meant by "G".'

Chapter 1

Background

The title of the thesis is "Using Scala for Computer-assisted Stepwise Refinement of Software (from Specification to Implementation)". This chapter introduces each of the concepts mentioned in this title, namely:

- computer-assisted activities and the manner in which instructions are formulated in preparation for computer processing, concentrating on grammars and syntax trees;
- Scala, which is a modern example of a programming language that incorporates features useful for translation between grammars;
- step-wise methods for arriving at a solution to a problem; and
- the production of software from specifications and implementations, including the definition of refinement.

1.0 Automation

The term "automation" refers to computer-assisted activities and the manner in which instructions are formulated in preparation for computer processing. With respect to this thesis, there are two distinct environments in which automation occurs. The first environment will be known as an interactive environment; the alternative is a facilitated environment. This distinction is relevant because the software developed in this thesis provides a facilitated environment by which to produce fully-verified software.

In the mode of operation referred to as an interactive environment, the operators direct the interactions between themselves and the computers. To do so, the operator expresses their instructions, typically by writing a program. The operator relies on an interpreter or compiler to instruct the machine to produce its results. Upon execution of the commands as encoded by the interpreter or compiler, the computer responds.

The alternative is a facilitated environment, in which the operator's direction is only over an intermediate client machine. The client machine provides indirect communication to a pool of servers which determine the response and report back to the client machine. The client machine in such an environment does not possess the algorithms required to solve the user's problem. Instead it has access to the target servers and a library of routines that allow it to translate the user's requests into the syntax of the target machine. The servers process the client's requests on their own schedule and respond only when the result is ready. In this thesis, the algorithms used by the server are said to belong to its oracles, which are often third-party components accessible only to the server.

```
0  a := b + a;
   b := a − b;
   a := a − b
```

Listing 1.0: Initial Document

```
0  declare t := a in (
   a := b
   b := t;
   )
```

Listing 1.1: Edited Document

The intent of the thesis is to use Scala to implement the server-side language so a client can check the users' progress as they edit code. As an example, consider that on the client computer a user opens a document containing the program given by Listing 1.0. After editing, the user ends up with the program seen in Listing 1.1.

The language discussed in this document allows the client software to formulate a query asking the server about any potential for failure introduced by the changes. The server facilitates this test by checking the query to determine which if any of its third-party oracles are able to process the query. When one of the oracles produces a response, the server informs the client which conveys the response to the user. In this way, the thesis describes a facilitated environment that can let a high-level source-code editor know how one version of a document stands in relation to another.

1.1 Grammars and Syntax Trees

This thesis describes a facilitated environment that can let a high-level source-code editor know how one version of a document stands in relation to another. To do

so, each version of the document must be expressible using the language known as SIMPPLE. Understanding the mechanics of this process requires a background in the fundamentals of high-level languages.

This section presents some of the fundamentals of high-level languages. The section begins by focusing on context-free grammars which define the language of the high-level instructions.[Chomsky, 1959] The discussion moves to syntax trees, which are the mechanism by which the instructions are prepared for semantic analysis and compilation/interpretation.

1.1.0 Context Free Grammars (CFGs)

Grammars consist of:

- a set of symbols (for example 1 and +); and
- syntactic production rules that combine the symbols.

The production rules used in this thesis allow symbols to be combined based on juxtaposition to other symbols. An example of combining symbols in such a manner is restricting the use of the symbol + so that it can only occur between two 1's. As an introductory example, rules of this nature are typically expressed in prose using statements such as:

- "1" is the name of a positive integer; and
- if "a" is the name of a positive integer, then "$1 + a$" is the name of a positive integer.

The grammar described above is expressive enough to represent all positive integers. Expressiveness does not imply practicality; a more practical grammar for representing integer values will be given later in this chapter.

The Backus-Naur form is conventionally used to write production rules precisely and succinctly. In this form, each production rule can be assigned a (not necessarily unique) label; the label is a nonterminal symbol in the grammar. These nonterminals symbols can be referred to by other production rules. The nonterminal symbol associated with the production rule is written on the left-hand side of an expression, and the string of terminal and nonterminal symbols on the right-hand side.

The BNF grammar equivalent to the prose example written above assigns the nonterminal label of $POSITIVE\_INT$ to both production rules. The second rule is recursively defined in the BNF grammar, yielding the two expressions below:

$$POSITIVE\_INT ::= 1$$

$$POSITIVE\_INT ::= 1 \quad + \quad POSITIVE\_INT$$

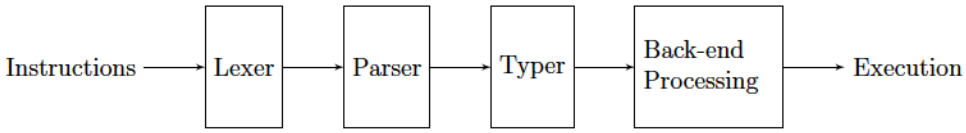
Extended BNF form allows rules to be combined using optional parts, alternatives, and other notational conveniences. The following extended BNF equation combines the two rules written above into one:

$$POSITIVE\_INT ::= 1 \quad [\quad + \quad POSITIVE\_INT \quad]$$

The limits on production rules used in this thesis ensure that the result is a context-free grammar (CFGs). CFGs and their extended BNF forms are readily parsable and implementable by computing machines. The ability to write extended BNF grammars therefore allows the design of machines that respond to commands tailored for specific data and processes.

1.1.1 Programming with CFGs

Commanding a machine using a CFG typically involves four processes:



- lexing

  In the lexing phase, character sequences are matched to regular expression descriptions of tokens.[Reilly *et al.*, 2000]

- parsing

  Parsing produces a parse tree from a string belonging to CFG.

- typing

  During typing, terminals and character data in the parse tree are converted to internal representations that are readily manipulated by the processor. This internal representation is referred to as a typed abstract syntax tree (AST); it is a more generalized (or abstract) representation of the parse tree than the original. The original is referred to as the concrete representation.

- back-end processing

  The final stage is to process the typed AST. Typically the typed AST is processed by a compiler-linker, interpreter or some other back-end process. In the case of an interpreter, the machine is instructed to perform a series of operations using the character data from the terminals as control values during this stage. A compiler-linker defers the execution of the instructions (with the

possibility of optimizations along the way) by encoding the typed AST in a native form.

1.1.1.0 Parse Trees

Parse trees are the mechanism by which the instructions defined by CFGs are prepared for semantic analysis and compilation/interpretation. Parse trees are acyclic rooted trees: they are described and understood using the terminology from graph theory. Parse trees can also be described in terms of the CFGs they represent. Every parent in the resulting tree corresponds to a nonterminal in the grammar. Every leaf corresponds to a terminal.

1.1.1.1 Abstract Syntax Trees (ASTs)

During typing, the input is converted to a typed AST and is then ready to be processed. The AST is a symbolic representation of the user's instructions. ASTs are abstracted or generalized versions of parse trees, and one AST may describe more than one parse tree. This feature of ASTs makes them especially useful in translating from one grammar to another.

In an interactive environment, processing takes an AST and provides a semantic interpretation for the expression represented. To do so, the processor must have access to algorithms expressed in machine language. As an example, the user may instruct the machine to compute the value of $\sin\pi$. Interpreting sin and π to be the trigonomic equivalents, a properly designed interpreter would compute the result 0.0; a compiler-linker would take a less direct route to the same end. In both cases, a machine-implementable algorithm for computing the sin of a number is required,

as is a finite interpretation of the constant π. The semantic interpretation is in most cases a practical one. For example we know the constant π to be infinite; a practical interpretation of π ensures that expressions containing π evaluate within an acceptable margin of error.

In a facilitated environment, which is the environment considered in this thesis, the AST is neither compiled nor interpreted, but serves as the basis for semantic analysis given a selection of back-ends. Semantic analysis matches the patterns and structures of the AST to machine-implementable algorithms that give meaning to the AST. In this thesis, these algorithms reside in the intellectual property of third-party programs at the back-end. Because of the arm's length relationship between the typer and the back-ends in these cases, it is helpful to think of the third-party programs as external oracles. In this situation, given an AST, automation involves one of two options:

0. if the oracle has an API, the AST is processed by translating it and passing it to the oracle for secondary processing using the API.

1. alternatively, the translator can work backwards from the translated AST to generate a concrete syntax for the oracle.

1.2 Scala: A Modern Workhorse

The Scala programming language is hailed as "the result of a careful integration of object-oriented and functional language concepts" [Odersky, 2014]. In this section, features of Scala that are especially well-suited to semantic analysis of ASTs are

```
0  i match {
       case 0 =>
                    0
       case nonzeroValue  if nonzeroValue > 0 =>
                    1
5      case  _  =>
                    −1
   }
```

Listing 1.2: Match-case Construction

introduced.

1.2.0 Match-Case Construction

In Scala, the match-case construction is used extensively to achieve pattern matching. The match-case construction consists of a match keyword preceded by an expression and followed by a block of code consisting of case statements. The case statement accepts an extractor, an optional guard, and a body of code which is executed if the expression is an application of the extractor.

An example of a match-case construction is presented in Listing 1.2. In a match-case construction, each extractor that immediately follows the case keyword attempts to represent the match expression as an application of the extractor. If a constant is used as an extractor, as in the first case statement of Listing 1.2, the only application is a variable whose value equals the constant; thus if the variable i evaluates to 0, this first case statement applies. When the case applies, control passes to the optional guard expression. The guard is a Boolean expression which may mention variables initialized by the match. If the guard evaluates to true, control passes to the body of the case and the match is complete. Otherwise control passes to the next case

```
0  someString match {
       case """0|[1-9] [0-9]*""".r =>
           println("The integer-valued string has no leading zeros");
       case """[0-9]+""".r =>
           println("The integer-valued string has leading zeros");
5      case _ =>
           println("The expression is not a valid integer-valued string");
   }
```

Listing 1.3: Regular Expression Matching

statement. A wildcard match in the form of "case _" is used to implement a default behaviour for the match. From this description, it should be understood that the example computes the sign of the integer valued variable i.

1.2.0.0 Regular Expressions and Implicitly-defined Extractors

Extractors are implicit in a number of cases. For example, Scala provides an augmentString class (strings delimited by three double-quotes) which can be used to convert the string such as $[0-9]+$ directly into a regular expression pattern matcher. By calling the method r of an augmentedString instance, a regular expression of type Regex is returned. Scala adds extractor support automatically to all regular expressions declared in the language. This means that in addition to the usual purpose of regular expressions for searching and replacing throughout a string, match-case constructions provides an additional mode of operation in which a single string is easily compared against any number of regular expressions.

This is demonstrated in Listing 1.3. In the example, a string is first compared against the regular expression $0\,|\,[1-9]\ [0-9]*$ and, failing that, is subsequently compared to $[0-9]+$. This sequence of comparisons captures integer strings of

decimal digits having no leading zeros first and processes them as a unique case of the more general expression.

1.2.0.1 Case Classes

A "case class" is a special form of class in which constructors double as extractors. An automatically generated unapply function is created that extracts an object's constructor arguments. [Subramaniam, 2009] In particular, each of the formal parameters of the case class constructor is assigned a place (a positional field) in the extractor's list of extracted values. Provided that the expression being matched is represented by an instance of the case class, variables associated with each place in the constructor pattern are in scope during execution of the guard expression and, in turn, the case body as would occur for any Scala extractor.

Case classes are especially helpful when working with ASTs. When using Scala to process ASTs created from CFGs, they make it easy to determine the specific production rule used to create each object. Root and child nodes can be passed as the input to a match-case construction to be compared against sub-trees of a specialized form. Once matched, the case classes provide access to the constructor arguments. In the case of ASTs, these constructor arguments are the terminals and nonterminals of the original BNF grammar. This greatly simplifies the typing, translation and processing of tree-like structures.

Case classes are typically used for the subclasses that extend a superclass. As a precaution against further changes to the number and type of subclasses supported by the grammar, the superclass can be marked as sealed. This forces all case classes to be declared in the same file as their superclass, and ensures that warnings will be

created if a match construct is not declared for each possible case class.

1.2.0.2 Advanced Extractors in Regular Expressions

Scala's support for regular expressions provides for support of capturing groups defined within the regular expression string. A RegEx instance whose regular expression has no parenthetic groups, such as $[0-9]+$ is matched against the extractor `matchDecimalDigits()` which accepts an empty set of arguments. For each group in the regular expression, a corresponding argument is added to the extractor. Characters outside of any group do not receive a place in the argument list. Optional and non-capturing groups are supported and can be nested within one another. Optional variables are assigned a place in the extractor, but their value can be null or the empty string. Non-capturing groups are not assigned a place in the argument list. Non-capturing groups nested inside capturing groups do not get a place of their own, but none-the-less are included in the string captured by its parent group.

As an example, the regular expression of the form $(-|\+)?0*([1-9][0-9]*|0)(?:(\.(?:[0-9]*[1-9])?)0*)?$ discards leading and trailing zeros in a decimal expansion. Furthermore, it separates the integer and fractional parts into separate groups, yielding the extractor prototype:

`matchDecimalExpansion(sign,wholePart,fractionalPart)`

By combining this feature with guard expressions in match-case constructions, separate case statements bodies can be written that target strings meeting very specific criteria.

1.2.1 Built-in Parser Combinators

Scala comes prepackaged with a utility library that includes parser combinators. The combinators extend the pattern matching ability of Scala to incorporate rules very similar to the BNF conventions. These utilities reside in the Scala package scala.util.parsing.combinator. To implement a parser combinator in Scala, one may extend one of the built-in Scala classes: RegexParsers, JavaTokenParsers, or PackratParsers.

1.2.2 Parametric Polymorphism

In order to define a function that can return generic types, the language supports parametric polymorphism. Functions and classes can have their return types and the types of their arguments parameterized by appending the function or class name with a type variable enclosed in square brackets. The type variable can be referred to during the declaration of the class or function. Instances of the class or function must define a type value to be substituted wherever the type variable appears. Constraints can be placed on the range of possible values that can be assigned. For example, the notation [N <: T] indicates it is possible to assign a type-value to N only if the type-value is derived from the type T.

1.2.3 Dealing With Failure: Try Monad and Option

The Try monad provided by Scala allows a return value of a function to be classified into success or failure. Case-class extractors allow a successful return value to be unapplied to return the result. On failure, it allows extra information to be provided

to the caller.

Scala also supports an Option type. Like the Try monad, it also allows a return value of a function to be classified into success or failure, using the term Some to represent success and None to represent failure. An Option type can be converted to a Try monad by converting a Some result to a Success result. A None result can be mapped to a Failure value.

1.3 Specification of Imperative Programs

A core theme throughout the thesis is the correct relationship between specifications and algorithms. This section introduces this theme through a discussion on specification of imperative programs.

1.3.0 Imperative Programs and States

The conventional description of an imperative programming language is that of a list of commands which are executed sequentially. The commands operate within a state space, and the effect of the command is to alter the state in a deterministic fashion. Logical questions can be asked about any two states. One such question returns what are known as program invariants; that question asks: given any pair of states modified by an imperative program, what statements are guaranteed to hold true. Another question returns what is referred to as the program specification: what relationship exists between the first and last state from a program sequence (from input to termination).

1.3.1 Programming Theory

Any cohesive axiomization of a language allowing its expression in predicative logic is generally referred to as a theory. Hoare's Logic is often referred to as the first usable theory of programming. Hoare introduced triples in the form P {Q} R as a notation meaning "If P is true before the initiation of Q, then the assertion R will be true upon its completion".[Hoare, 1969] The logic is composed of: a) an infinite set of axioms described by an axiom schema for assignment; and b) rules of consequence, composition, iteration allowing new theorems to be deduced from axioms and theorems already proved. The resulting system tackles the problem of correctness of an algorithm.

Building on the concept of precondition and postcondition, Dijkstra[Dijkstra, 1975] developed a formally rigorous calculus allowing preconditions to be determined from programs and post-conditions. This calculus was especially geared towards the formal derivation of programs. A more practical (or pragmatic) approach was proposed by Hehner in 1984. Hehner's efforts turned questions about sequences of states into a method for program development with proof at each step. Hehner took the view that "[a] proof can be a continuing equation, ... it can also be a continuing implication, or a continuing mixture of equations and implications." [Hehner, 2014]. Because programs are likewise written as sequences of steps, this approach makes it "easier to see the execution pattern when we retain only enough information for execution." [Hehner, 2014].

Hehner combined the pre-conditions and post-conditions into a single Boolean formula. Programs are a distinguished subset of these specifications. Specifically, "A

program is a specification of computer behavior; for now, that means it is a Boolean expression relating prestate and poststate. Not every specification is a program. A program is an implemented specification, that is, a specification for which an implementation has been provided, so that a computer can execute it." [Hehner, 2014] Formal derivation of programs in Hehner's approach is accomplished by reducing a specification to this distinguished subset while using reverse implication to ensure correctness. This approach is referred to as stepwise refinement.

1.3.2 Interpretations of Commands and Refinement

Theories of predicative programming and programming by specification were developed and well-understood by the mid-1990's. These theories come in a number of forms. A comprehensive theory of programming by refinement is actively maintained by Hehner[Hehner, 2014] and the reader is referred there for a guide to the strategies by which a program is developed from its specification. Hehner's work includes contributions from Norvell, especially with regard to theories for nondeterminism in functional programming and for function refinement, the recursive definition of while-loops as well as criterion for data transformers.[Hehner, 2014]

Hehner adopts a first-order form that views specifications as Boolean formulas and refinement as ordinary implication. A closely related higher-order model views specifications as a function of two states, and refinement as a preorder[0] between specifications. This distinction is abstracted away when discussing programming instructions, but becomes important when formulating specifications. As an example the theories included in a NECEC 2010 conference paper[Motty and Norvell, 2010]

[0] a preorder is a binary relation that is reflexive and transitive

are reproduced in Table 1.0.

Term	First-order Interpretation	Higher-order Interpretation
$\|\|\mathbf{skip}\|\|$	$x' = x \wedge y' = y$	$\lambda\, s \cdot \lambda\, s' \cdot (s = s')$
$\|\|E\|\|$	E	$\lambda\, s \cdot E^{x,y}_{sx,sy}$
$\|\|x{:=}E\|\|$	$x' = E \wedge y' = y$	$\lambda\, s \cdot \lambda\, s' \cdot \forall v \cdot\ s'\, v = \|\|E\|\|\, s$ $\lhd v = x \rhd s'\, v = s\, v$
$\|\|\langle S \rangle\|\|$	S	$\lambda\, s \cdot \lambda\, s' \cdot S^{x,y,x',y'}_{sx,sy,s'x,s'y}$
$\left\|\begin{array}{l}\mathbf{if}\ E \\ \mathbf{then}\ F \\ \mathbf{else}\ G\end{array}\right\|$	$\|\|F\|\| \lhd E \rhd \|\|G\|\|$	$\lambda\, s \cdot \lambda\, s' \cdot \|\|F\|\|\, s\, s' \lhd \|\|E\|\|\ s \rhd \|\|G\|\|\ s\, s'$
$\|\|F; G\|\|$	$\exists \dot{x}, \dot{y} \cdot \|\|F\|\|^{x',y'}_{\dot{x},\dot{y}} \wedge \|\|G\|\|^{x,y}_{\dot{x},\dot{y}}$	$\lambda\, s \cdot \lambda\, s' \cdot \exists \dot{s} \cdot \|\|F\|\|\ s\, \dot{s} \wedge \|\|G\|\|\ \dot{s}\, s'$
$\|\|F \implies G\|\|$	$\|\|F\|\| \Rightarrow \|\|G\|\|$	$\lambda\, s \cdot \lambda\, s' \cdot \|\|F\|\|\ s\, s' \Rightarrow \|\|G\|\|\ s\, s'$
$\|\|F \sqsubseteq G\|\|$	$\forall x, y, x', y' \cdot \|\|G\|\| \Rightarrow \|\|F\|\|$	$\forall s \cdot \forall s' \cdot \|\|G\|\|\ s\, s' \Rightarrow \|\|F\|\|\ s\, s'$

Table 1.0: Interpretations of Commands and Refinement[1]

1.3.3 Choice of State Function

The term "state space" refers to the abstract mathematical universe whose axes are the state variables of a program. A state is any point in the state space. As such, a state maps variable names to values (the "coordinates" of the state). A state function is an abstract mathematical concept that accepts a state and a state variable name and returns the corresponding value of the state variable in that state. The details of how the state function is defined is important to this thesis.

In chapters 2 to 7 of this thesis, the state function maps the state to a distinct and unique set of global variables. This definition of a state function results in a first-order model of programs. The disadvantage is that some statements about programs

[1] In Table 1.0, it is assumed that all commands operate on states whose state variables include only x, and y. Formulas need to be modified accordingly for other states.

are inexpressible in first-order logic. An implementation plan to provide an higher-order state function is presented in Chapter 8 of this thesis. The higher-order state function treats states as functions from state-variable names to dependently-typed values. So in Chapter 8, if s is a state, it is also a dependently-typed function. The type of the function is dependent on its argument; for example, if x is the name of an integer-valued state variable, while y is the name of a Boolean-valued state variable, then $s(x)$ is an integer value, and $s(y)$ is a Boolean value.

1.3.3.0 Representing Programs using Higher-order Logic

The higher-order axioms, theorems and definitions of programming theory used in this thesis are adapted from Norvell[Norvell, 2012]. According to Norvell's method[Norvell, 2012], specifications and programs are functions; they accept two states and return a Boolean value, and can be compared to one another provided higher-order typing rules are respected. Traditionally an unprimed variable is used to represent the source state, and a primed variable represents the target state. Each state has a signature, Σ defined as a set of pairs (v, T), where v, a string, is a variable name and T is the type of data associated with the named variable.

Using this notation, the programmer can enter the specification and assert that it has Boolean relationships with other specifications. Refinement (definition 1.0) is a higher-order relationship which compares two specifications. Refinement uses implication to partition a set of programs into those that meet and those that do not meet a specification. The definition of refinement introduces the symbol $\sqsubseteq$, pronounced "is refined by". So the term $S \sqsubseteq P$ is pronounced "S is refined by P", or equivalently "P refines S".

Definition 1.0 [2] $S \sqsubseteq P \equiv (\forall_{\Sigma} s, s' \cdot Sss' \Longleftarrow Pss')$

Since logical implication over sentences is a preorder, refinement is too. Two special specifications mark the floor and ceiling of the refinement preorder. First there is the case of a specification that is tautologically true and therefore holds for all pairs of states. As an example, the Boolean expression $\top$ is a constant that always evaluates to true. A lambda abstraction[3] can be used to turn this constant into a function that accepts two states s and s' as input but subsequently always returns $\top$. The resulting specification (see definition 1.1) is given the name 'abort': such a specification is always satisfied.

Definition 1.1 $abort \equiv \lambda_{\Sigma} s, s' \cdot \top$

On the contrary, it is possible to write a specification that is unsatisfiable and therefore cannot hold for any pair of states. This specification (see definition 1.2) is given the name 'magic'; its definition is the lambda abstraction which evaluates in all cases to false:

Definition 1.2 $magic \equiv \lambda_{\Sigma} s, s' \cdot \bot$

The special names of the specifications are chosen due to the contractual nature of specifications. For instance, it would be unrealistic to search for a specification equivalent to 'magic': no program can satisfy a search for the impossible. On the contrary, a search for a program that satisfies the specification 'abort' will always

[2] The symbol $\forall_{\Sigma}$ is used here to mean universal quantification over objects of type Σ, where Σ is the type for states

[3] The symbol λ_{Σ} is used here to mean lambda abstraction over objects of type Σ, where Σ is the type for states

succeed; there's just no point running it since it does nothing useful towards the requirements. Refinement is a preordering relation on specifications. At the bottom of this preorder sits the specification known as abort. All specifications refine abort, but abort refines no other specification but itself. At the top of the preorder sits magic. No specification but magic refines magic, though magic refines all specifications. This can be summarized notationally[4] as:

$$\forall_{\Sigma \to \Sigma \to Bool} S \cdot abort \sqsubseteq S \sqsubseteq magic$$

In between the two extremes, the ability to partition a set of programs into those that meet a specification and those that do not is critical to the software acceptance and development process. Refinement is transitive: "...if you refine a program today, and refine it further tomorrow and again the day after, then by the weekend you still have a refinement of what you started with."[Morgan, 2009]. Since every specification is refined by at least one unimplementable specification (i.e. magic), implementability is a critical threshold. "[A]n unimplementable specification can not be refined by an implementable specification."[Norvell, 2012]

1.3.3.1 Representing Programs using First-order Logic

Unlike higher-order logic, in first-order logic, well-formedness rules of propositions do not allow relationships between functions. In first-order logic, Boolean-valued terms are predicates of a known arity; a predicate flanked by the appropriate number of inputs evaluates to either true or false. Predicates must be fully applied to be well-

[4]The symbol $\forall_{\Sigma \to \Sigma \to Bool}$ is used here to mean universal quantification over objects of type $\Sigma \to \Sigma \to Bool$, where Σ is the type for states

formed.[Wang, 1961] Consider for example that we have two specifications, $S1$ and $S2$, such that two states s and s' satisfy one specification if and only if they satisfy the other. In first-order logic, this is written[5] as:

$$\forall_{\Sigma} s, s' \cdot S1ss' \iff S2ss'$$

The rules of first-order logic allow one to conclude that $S1$ and $S2$ are functionally equivalent, but fall short of being able to conclude that they are equal; that is: $S1 = S2$. To do so would be to write predicates $S1$ and $S2$ without arguments, which would be a violation of the well-formedness rules.

1.4 Summary

Those who wish to use a computer to automate a process often rely on custom languages defined by formal grammars to express a goal, command, or instruction. Modern-day programming languages such as Scala contain features that simplify the parsing of such grammars. In this thesis, a custom language built to express concepts from programming theory is parsed and processed by a server written in the modern-day programming language Scala.

The topics presented in this chapter serve to introduce basics of grammars, the Scala language and programming theory. The chapters which follow build the grammar that the server software will use to record and represent documents to which programming theory applies.

[5] The symbol $\forall_{\Sigma}$ is used here to mean universal quantification over objects of type Σ, where Σ is the type for states

Chapter 2

SIMPpLE and its Abstract Syntax

In this section, past developments and background work completed prior to the commencement of this thesis is discussed. This work includes the abstract syntax of SIMPpLE.

2.0 SIMPpLE Theory of Programming

It is well known that syntactically and behaviourally correct software implementations can be derived directly from specifications. The process known as stepwise refinement derives the program from a specification and verifies it as it is constructed line by line.

The need for an editing and proving environment in which programs are treated as objects of enquiry led to the creation in 2007 of the SIMPpLE project. SIMPpLE is an acronym for Structured Imperative Modular Proof/Programming Language and Environment. The aim of the SIMPpLE project is to design a formal language for writing program specifications, programs, and proofs that programs meet their speci-

fications. [Norvell, 2009] SIMPPLE provides a strongly typed syntax for interspersing specifications with commands. Specifically, it provides a grammar for recording:

- arithmetic and logical expressions with precedence and associativity
- specifications
- commands (including assignment, sequential composition and alternation)
- lambda notation
- theorems
- derivations of programs

2.1 Client/Server Data Flow

Early work on SIMPPLE produced the client-side editor: a browser-based extension that allows users to edit proofs. The client-side editor is written in JavaScript targeting the Mozilla framework. Behind the scenes, the editor communicates using structured XML to a verification host.

It is the intent that the host will use the editor's XML requests to maintaining an abstract representation of the user's document in memory on the server-side host. This abstract representation takes the form of an AST: a hierarchy of Scala objects and classes. The server analyzes these ASTs to generate queries that can be posed to one or more automated theorem provers to test the correctness of the program and any assertions made about it. This data flow is depicted in Figure 2.0.

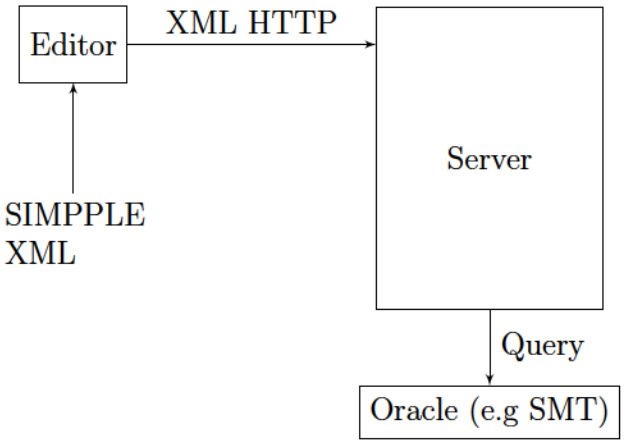


Figure 2.0: Client/Server Data Flow

In this section, a demonstration of stepwise refinement of a specification is introduced. The demonstration is visualized using a mock-up of the client-side editor, and shows how such an editor would assist in the stepwise refinement of software from specification to implementation. The demonstration here is intended to provide an initial introduction to the process. The demonstration is discussed further later in the thesis once additional program theory and an abstract and concrete syntax for the server-side language has been presented.

A comprehensive theory of programming by refinement is actively maintained by Hehner[Hehner, 2014] and the reader is referred there for a guide to the strategies by which a program is developed from its specification. In the interest of providing an example, however, it is assumed that some approaches to mathematical problem solving are universal. Of these, the formulation of the problem statement, and the solution of trivial cases are sufficient for demonstration purposes.

2.1.0 Formulating a Specification

The first step in the practice of stepwise refinement of software is to formulate a specification. While this may sound time-intensive, the approach suggested in this demonstration is to treat functions as uninterpreted constants, or free variables. Uninterpreted function names, when encountered in a document, are treated as constants in the AST. Using constants for these names reinforces that the name refers to an object with universal, immutable characteristics. This allows the writer to freely introduce a new function or concept as a placeholder for something to be defined more fully later. The theory behind uninterpreted functions is beyond the scope of the thesis, but comes from the study of many-sorted logic, a logic system widely used in modern SMT solvers.[Barrett *et al.*, 2010]

To demonstrate, the problem of computing the greatest common divisor [Norvell, 2012] has been adapted in this thesis for use as an example in the following sections.

2.1.0.0 Defining Input and Output Relationships

In Proposition 2 of Book VII of the Elements, Euclid proposed "Given two numbers not prime to one another, to find their greatest common measure." [Hawking, 2005] This statement on its own defines the nature of the problem to be solved. On the given, or input side, two numbers are presented. Provided that the preconditions are met, the algorithm is to produce the greatest common measure of the two; otherwise there is no requirement.

Euclid's second proposition is typical of many computing problems in that it defines constraints on the problem that must hold in order for the algorithm to apply.

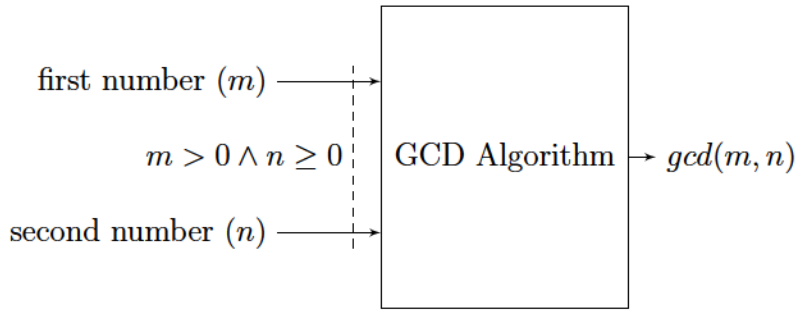


Figure 2.1: GCD Blackbox Function

In Euclid's case, the constraint is that the inputs must not be prime to each other. It's worth noting that the Euclid's algorithm actually works in situations where the inputs are prime to one another; the resulting GCD in that case is unity. Modern versions of the algorithm generally weaken the precondition, requiring only that at least one of the inputs be non-zero. The block diagram in Figure 2.1 shows this relationship in graphical form.

When writing input/output specifications, it is common to reuse state-variable names to hold both the input and the output values. Conforming to this practice, and using the uninterpreted function name gcd to represent the central block, the specification for Figure 2.1 corresponds to the input string:

$$\langle m > 0 \wedge n \geq 0 \Rightarrow m' = \mathrm{gcd}(m, n) \rangle$$

2.1.0.1 Elements of the Editing Environment

Once an input and output relationship has been defined, it can be entered in a custom, web-enabled editor for version control and analysis. The editing environment requires a number of elements, as illustrated by the mock-up in Figure 2.2. The mock-up

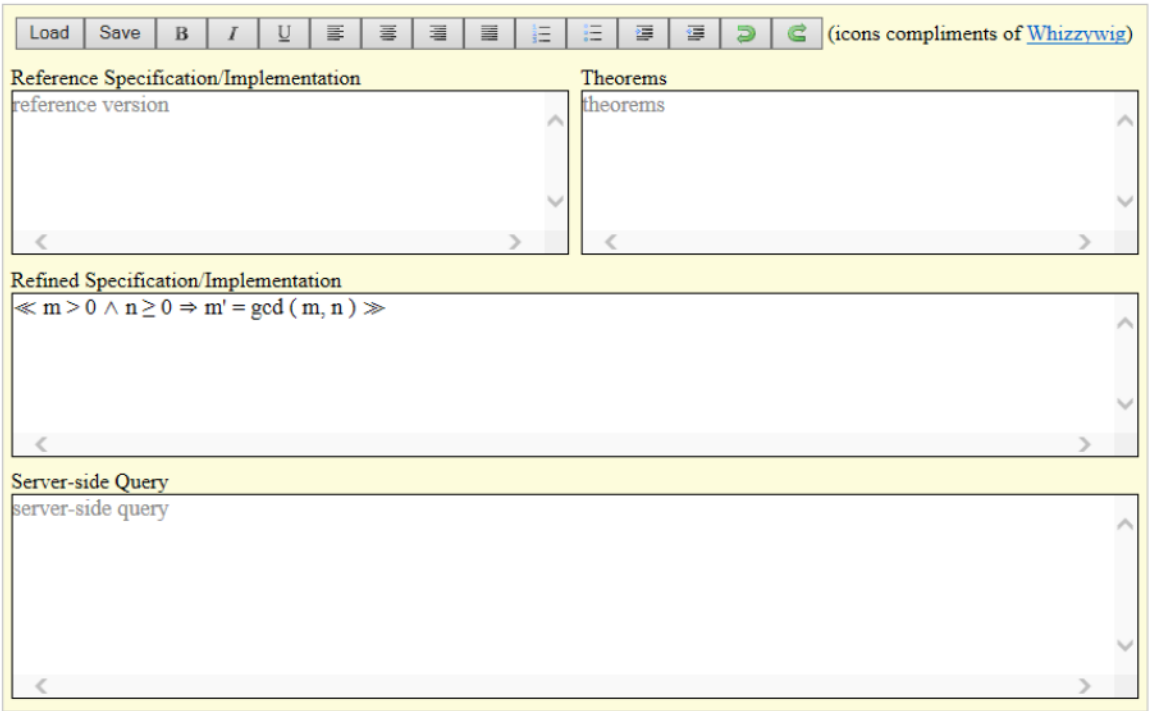


Figure 2.2: Input/Output Relationship in Editor Mockup

allows a previous revision of the document to serve as a previous step in the refinement process. The active document forms the current step. Theorems justifying the revision can be entered and maintained separately by the user. As the user edits the document, the editor formulates a query to pose to the server-side prover to check the correctness of the user's modifications. A read-only window allows the display of the server-side query for diagnostics and development purposes.

2.1.1 Trivial Solution and Other Theorems

Many problems have a set of known solutions that can be easily solved. In the case of the GCD, a trivial solution occurs when one of the inputs is 0 and the other is a positive number. Since zero can be divided by any number, the largest number

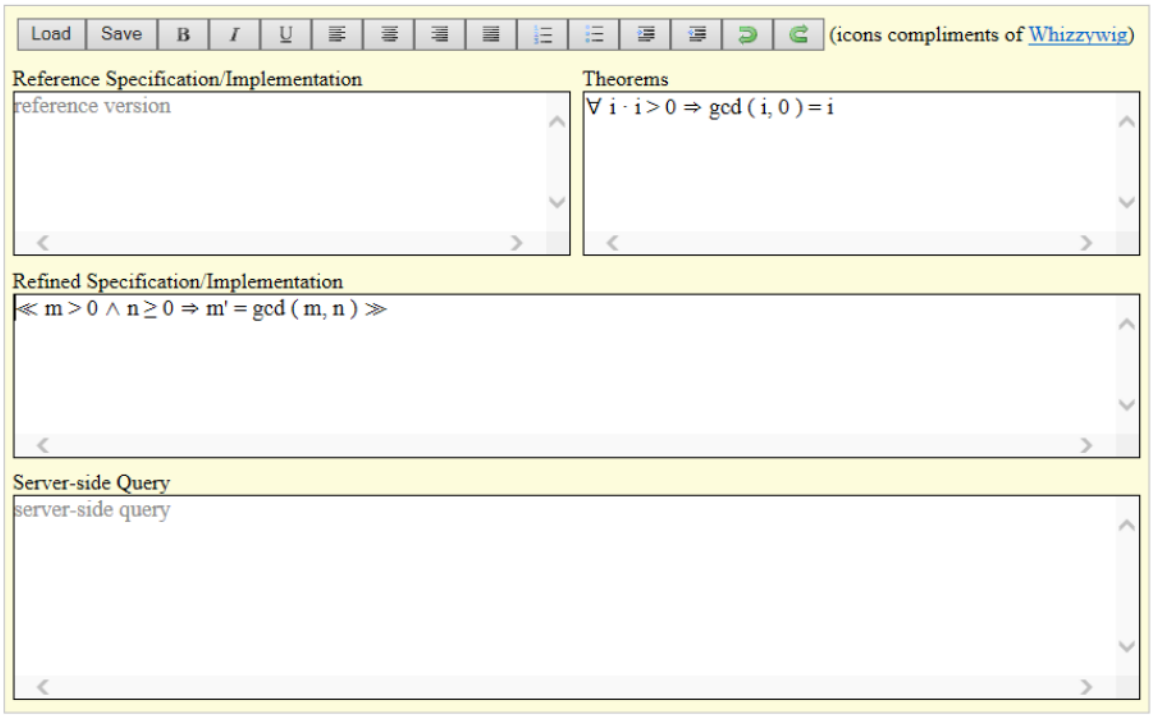


Figure 2.3: Axiom Introduction in Editor Mockup

which divides the other number is therefore the solution. Since the other number is positive number it divides itself and is the largest number to do so; that makes it the solution. We can express this logically as Theorem 2.0. This theorem can be added to the document as illustrated in Figure 2.3.

Theorem 2.0 $\forall i \cdot i > 0 \Rightarrow \gcd(i, 0) = i$

2.1.2 Application of Rules

Given an input and output specification and a trivial solution, the next step is to write a program that solves the trivial case without changing the overall specification of the problem. It is at this stage that the use of formal methods diverges from approaches generally followed by programmers. The traditional approach used would

be to implement the trivial case and make the software bullet-proof so that it aborts, retries, or ignores in every other situation. The method used here is to implement the cases where it can be proven that a solution exists and abort only if it can be proven that one solution is as good as any other.

A fine-grained formal proof is a human-verifiable proof that must be applied one rule at a time. Rules allow expressions of a language to be rewritten in a different form without altering its meaning. For example, the case creation law[Hehner, 2014] states that for any specification P, and any boolean expression b, P can be rewritten as $\mathit{if}\ b\ \mathit{then}\ \ b \Rightarrow P\ \mathit{else}\ \ \neg b \Rightarrow P$. SIMPPLE uses the closely related Alternation law, in which angle brackets convert the boolean expression to the higher-order type of a specification; hence one writes $\mathit{if}\ b\ \mathit{then}\ \langle\ b\ \rangle \Rightarrow P\ \mathit{else}\ \langle\ \neg b\ \rangle \Rightarrow P$.

Often in conventional formal proofs, rules are combined into one step and the verification of proper application of the rules is left to the motivated reader. SIMPPLE allows derivations using either a fine-grained formal proof or a conventional formal proof. Automated verification of proofs is provided by the server-side oracles.

Later in this thesis, this example will be continued by applying rules of programming theory to write an implementation for the trivial case without altering the specification as a whole. We will also introduce a recursive method for solving the non-trivial part of the problem, along with theories that justify the proposed implementation.

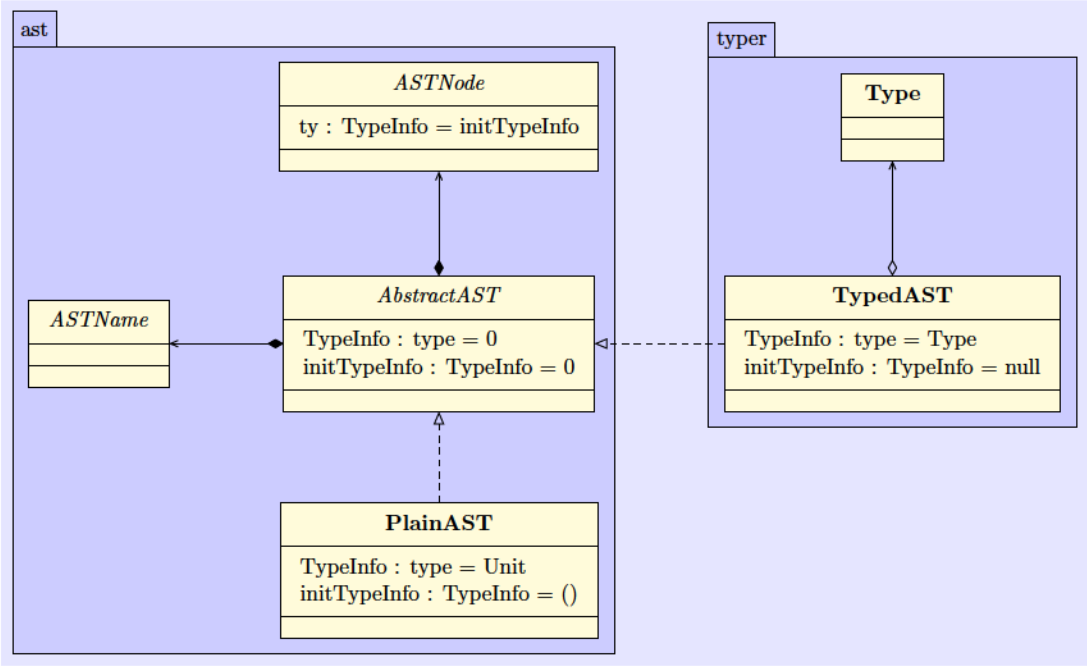


Figure 2.4: Class Diagram for AbstractAST

2.2 Abstract Syntax

An abstract representation of the user's document is maintained in memory on the server-side, with updates managed by HTTP communication between the server and client. The abstract representation takes the form of an AST: a hierarchy of Scala objects and classes. The AST and its type system were prototyped in Haskell and ported to Scala prior to the commencement of the thesis. Figure 2.4 shows the collection of Scala classes that are used in this thesis to represent the system's input and intermediate formulae on the server-side. These classes can be described by the abstract grammar presented in this section. This grammar allows the developer to write specifications and implementations interchangeably within the context of one document.

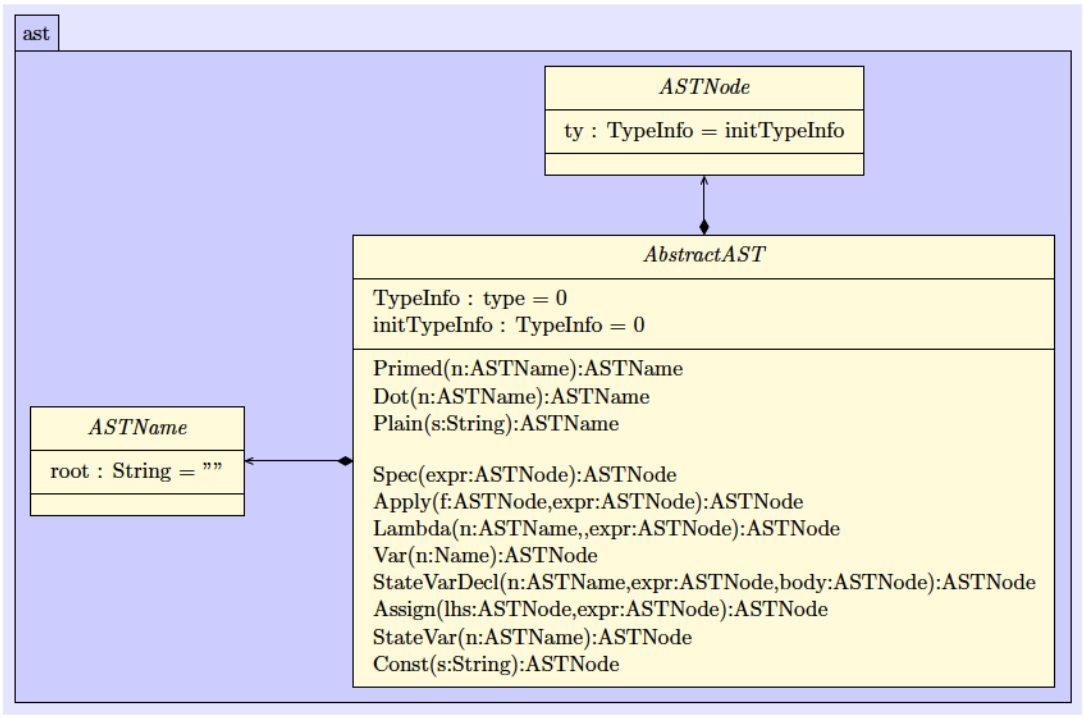


Figure 2.5: ASTNames and ASTNodes

ASTNodes and ASTNames are generated from the input strings using object-oriented techniques. A class diagram showing the internal methods defined by ASTNodes and ASTNames is presented in Figure 2.5.

In Scala, ASTName and ASTNode are declared as sealed abstract classes: each of the non-terminals that produce them are implemented as case-classes. The AST case-classes in Table 2.0 implement the AST used for server-side representation of a SIMPPLE document.

While the case classes of the AST given provides a summary of the elements representable within the AST, it does not provide a full description of the purpose of each element. Additional details are provided in Chapter 3. To facilitate the discussion, a concrete syntax is defined in Chapter 3 to illustrate various input strings

accepted by the server.

$Spec(expr : ASTNode) : ASTNode$
$Apply(f\text{: } ASTNode, expr : ASTNode) \ : ASTNode$
$Lambda(n\text{: } ASTName, expr : ASTNode) \ : ASTNode$
$Var(n\text{: } Name) \ : ASTNode$
$StateVarDecl(n\text{: } ASTName, expr : ASTNode, body : ASTNode) \ : ASTNode$
$Assign(lhs : ASTNode, expr : ASTNode) \ : ASTNode$
$StateVar(n\text{: } ASTName) \ : ASTNode$
$Primed(n\text{: } ASTName) \ : ASTName$
$Dot(n\text{: } ASTName) \ : ASTName$
$Plain(s\text{: } String) \ : ASTName$
$Const(s\text{: } String) \ : ASTNode$

Table 2.0: AST Case Classes

2.2.0 PlainAST: A Simply-typed Typed AST

The case class productions above are implemented within the context of the AbstractAST trait. The AbstractAST trait is an abstract trait in Scala and serves as a namespace within which ASTName and ASTNode can be parameterized. The parameterization is accomplished by declaring (but not setting) variables and types; these variables and types must be initialized by a derived concrete trait which is said to be the typed ASTs. Type information is attached to the ASTNodes by the typed AST. This relationship is depicted in Figure 2.6.

In order to represent ASTs prior to typing, the trait PlainAST is derived from AbstractAST. PlainAST defines TypeInfo as a Scala Unit. The default value of initTypeInfo is defined by PlainAST to be a 0-tuple (or empty tuple).

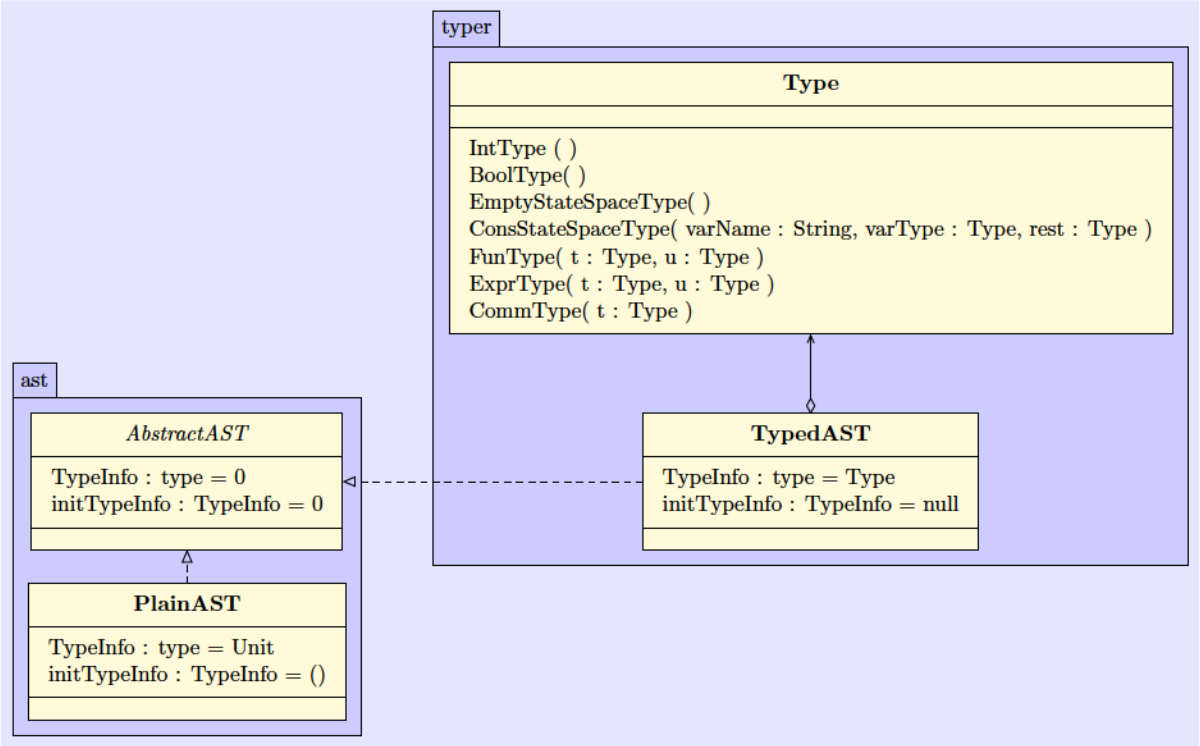


Figure 2.6: Concrete (i.e. Typed) ASTs

2.2.1 TypedAST: A Fully-typed Typed AST

The AST described above is very expressive, especially when it has type information assigned to each node. The type-system used for the majority of SIMPPLE documents was originally developed in Haskell [Norvell, 2009]. The implementation has been ported to Scala and is encapsulated by the TypedAST trait which derives its AST from AbstractAST as seen in Figure 2.6. The type system includes support for type constraints, type variables, and type constants, the full details of which are beyond the scope of the thesis. Of interest to the thesis are the type constants defined by the type system. Table 2.1 lists the most commonly used Type constants for the TypedAST trait.

In this thesis, a SIMPPLE document consists of a collection of theorems which represent statements made about a program and the behaviours its user expects of it. The types defined in Table 2.1 are sufficient to identify programs and specifications within the representative ASTs.

FunType (*t*: *Type*,*u*: *Type*)
ExprType (*t*: *Type*,*u*: *Type*)
CommType (*t*: *Type*)
IntType ()
BoolType ()
ConsStateSpaceType (*s*: *String*,*varType*: *Type*,*rest*: *Type*)
EmptyStateSpaceType ()

Table 2.1: Type Constants for the TypedAST Trait

2.3 Summary

SIMPPLE documents combine program specifications, programs, and proofs that programs meet their specifications. The server software for SIMPPLE uses the abstract syntax described in this chapter to record and represent SIMPPLE documents as a collection of ASTNodes. This allows the documents to be treated as objects of enquiry. The following chapters provide details on how client software can pose questions to the server in response to a direct query or in response to changes made to the document. It will be seen that also built into the server are theories of programming as defined for assignment, application and every other ASTNode.

Chapter 3

Converting Input Strings to ASTs

The intent of the thesis is to use Scala to implement the server-side language so a client can check the users' progress as they edit code. As mentioned in Chapter 1, consider that on the client computer a user opens a document containing the program given by Listing 3.0. After editing, the user ends up with the program seen in Listing 3.1

The language discussed in this document allows the client software to formulate a query asking the server if the modified file is a refinement of the original. The query must be expressible as an AST as described in the previous chapter. This chapter describes the mechanics of this process in detail through the high-level description of Scala source-code that converts server-side strings into the equivalent AST.

3.0 Concrete Syntax

While the syntax of the AST given in the preceding chapter provides a summary of the elements representable within the AST, it does not provide a full description of

```
0  a := b + a;
   b := a − b;
   a := a − b
```

Listing 3.0: Initial Document

```
0  declare t := a in (
   a := b
   b := t;
   )
```

Listing 3.1: Edited Document

the purpose of each element. Additional details are provided in this section. To facilitate the discussion, a concrete syntax is defined to illustrate various input strings accepted by the server.

The concrete syntax is produced by defining a BNF grammar for the terminals abstracted away by the AST. Parser combinators are used to collect the elements of each string and produce an equivalent AST for digital storage. The grammar also adds a number of pre-defined constructs and built-in operators necessary to formulate easy to read strings that include boolean, arithmetic and programming theory. Support for infix notation, tuple-form functions and additional tokens is added where necessary to improve readability of the input strings.

3.0.0 Overview

At the highest level, a document in SIMPPLE is a *TREE* about which queries can be made. Each query is a collection of *TREE*s. The concrete syntax for a query is given as:

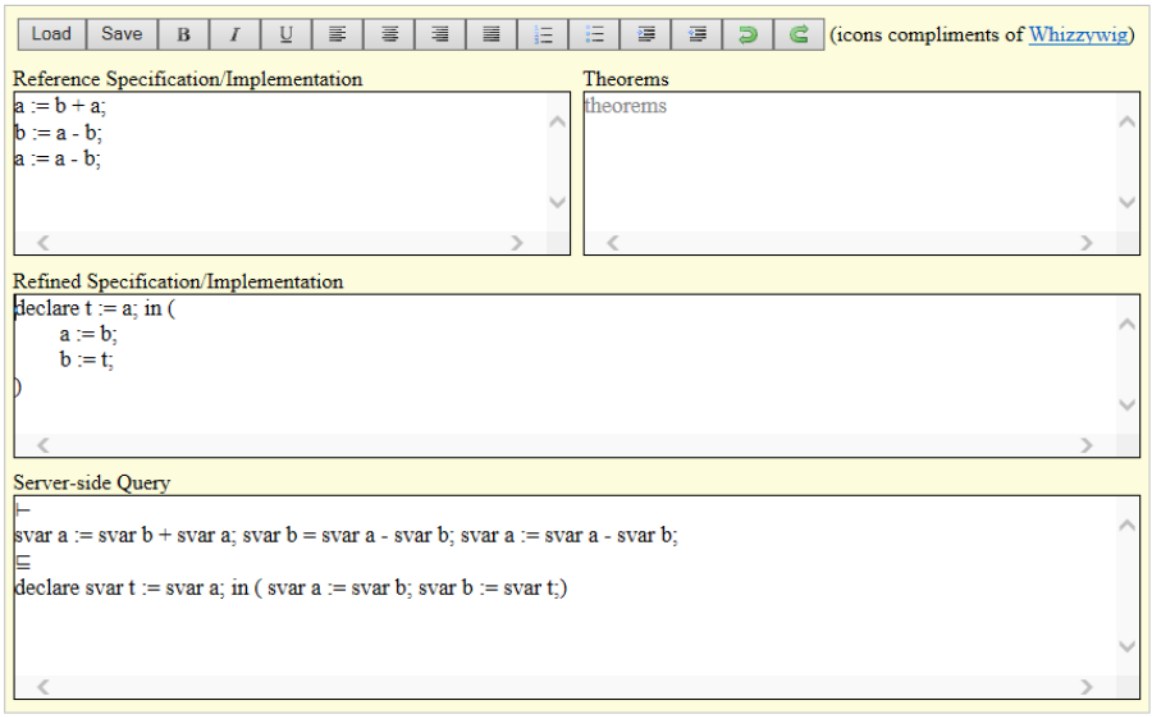


Figure 3.0: Server-Side Query in Editor Mockup

$$QUERY ::= \{ \vdash \ TREE \ \{ \ \sqsubseteq TREE \ \} \ \}$$

An example of a server-side query can be seen in Figure 3.0.

3.0.0.0 Queries

Each $TREE$ in a query serves as a theorem with an optional proof. If the theorem has no associated proof it is treated as a constraint on the system and given special consideration during processing of the query. Theorems introduce new terms and functions and define them by assertions and constraints. If a proof is provided, the $QUERY$ is referred to as a derivation. The only proofs considered in the thesis are those in which the conclusion can be reached through continued refinement of the initial tree. A modification to an existing document that progresses a specification

towards an implementation can also be modelled as a continued refinement of the initial document.

While not a formal part of the AST, theorems and derivations can be represented in abstract form as sequences of ASTNodes. Every time a new theorem without a proof is encountered, its AST is added to the list of theorems. As for proofs, an AST is produced for all consecutive pairs of *TREE*s in the derivations, relating the first *TREE* to the second. Each AST is added to the list of derivations.

For emphasis, the theorems considered in this thesis are classified as either a declarative assertion or a stepwise, equational proof. The proof, if included, must be a sequence of refinements; in this case the theorem proved is that the first *TREE* is refined by the last.

Declarative assertions are those theorems that have no associated proof. Theorems without proofs may be introduced at any point in a document but will be given special consideration during processing of the query. The system treats declarative assertions as either a constraint, axiom or definition.

Derivations are theorems with proofs. In this thesis, derivations are in the form of continued refinement. Derivations are checked for correctness by the system. Built into the server are theories of programming as defined for assignment, and application of some predefined operations discussed later in this chapter. In this way, each query provides a means for client software to pose questions to the server in support of validating revisions being made by the user to a document in its editor.

```
0  ⊢
   svar a := svar b + svar a;
   svar b := svar a − svar b;
   svar a := svar a − svar b
   ⊑
5  declare svar t := svar a in (
       svar a := svar b;
       svar b := svar t
   )
```

Listing 3.2: Server-side Query

3.0.0.1 Trees

In the existing implementation, each of the trees in a query is converted independently. Each *TREE* is an AST as represented by its root *ASTNode*. A *TREE* represents either an axiom, a specification, a command or an entire program. Given two *TREE*s, the client-side editor generates a *QUERY* asking whether the refinement between the left-hand side and the right-hand side is true. Listing 3.2 shows the query that corresponds to the document depicted in Figure 3.0.

The *TREEs* considered in this thesis are defined such that they always produce a *BLOCK* whose AST has type *CommType*(*t*: *Type*) where *t* represents the state space upon which the command operates. Trees consist of a single block, but optionally support a lifted form of implication informally referred to as "Leads to" which makes the evaluation of the second *BLOCK* conditional on the first. Because SIMPPLE is oriented towards imperative programming, *BLOCKs* combine statements to form an imperative path of execution. Each statement is separated by a semicolon.

TREE ::= *BLOCK* [=> *BLOCK*]

BLOCK ::= *STATEMENT* { ; {; } [*STATEMENT*] }

3.0.0.2 Statements

Like trees, *STATEMENTs* considered in this thesis always produce an AST of type *CommType*(*t*: *Type*). Statements include commands and specifications including nested *TREEs* but excluding recursive for and while loops. Recursive commands can be handled through a series of proofs. [Norvell, 2012] The treatment of recursion involving a proof that a monotonically decreasing sequence converges on a solution set for a problem is demonstrated later in this thesis through a case study on the derivation of Euclid's GCD algorithm.

$$
\begin{array}{l}
\mathit{STATEMENT} ::= \left(\begin{array}{ll}
\mathit{skip} & | \\
\mathit{BRANCHSTATEMENT} & | \\
\mathit{NESTEDTREE} & | \\
\mathit{SPECSTATEMENT} & | \\
\mathit{DECLARESTATEMENT} & | \\
\mathit{ASSIGNSTATEMENT} &
\end{array} \right) \\
\mathit{BRANCHSTATEMENT} ::= \quad \mathit{if}\ \mathit{CMDEXP}\ \mathit{then}\ \mathit{STATEMENT} \\
\qquad\qquad\qquad\qquad\qquad [\ \mathit{else}\ \mathit{STATEMENT}\ ] \\
\mathit{NESTEDTREE} ::= \text{`('}\ \mathit{TREE}\ \text{`)'} \\
\mathit{SPECSTATEMENT} ::= \ \mathit{UGSPEC} \mid \mathit{EQSPEC} \mid \text{`}\langle\text{'}\ \mathit{EXP}\ \text{`}\rangle\text{'} \\
\mathit{UGSPEC} ::= \ \forall\ \mathit{NAME}\ \cdot (\ \mathit{UGSPEC} \mid \mathit{EQSPEC} \mid \ \mathit{EXP}\ ) \\
\mathit{EQSPEC} ::= \ \exists\ \mathit{NAME}\ \cdot (\ \mathit{UGSPEC} \mid \mathit{EQSPEC} \mid \ \mathit{EXP}\ ) \\
\mathit{DECLARESTATEMENT} ::= \quad \mathit{declare}\ \mathit{PLAINNAME}\ := \mathit{EXP} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{in}\ \mathit{STATEMENT} \\
\mathit{ASSIGNSTATEMENT} ::= [\mathit{assign}]\ \ [\mathit{svar}]\ \ \mathit{PLAINNAME}\ := \mathit{EXP}
\end{array}
$$

Table 3.0: Abstract Syntax of Statements

In the concrete syntax for *STATEMENT*s, *DECLARESTATEMENT* and *ASSIGNSTATEMENT* only operate on state variables, so the name of the target is implicitly a state variable and need not be explicitly stated. Also, to further

reduce redundant terms in the examples, the frequently used keyword "*assign*" is optional.

3.0.0.3 Expressions

The concrete grammar supports the creation of terms and expressions, including infix operations with left and right associativity, and functions that act on tuples. The grammar ensures the correct precedence of standard arithmetic operations including multiplication and division, and also honors left and right associativity of the operators. Additionally, the grammar and its implementation supports parentheses in expressions used by both specifications and commands. Functions, including the predefined operations (addition, subtraction, multiplication, division and remainders) are converted to *CONST* and applied to their arguments accordingly.

$$
\begin{array}{l}
EXP ::= \ OP\_(0) \\
OP\_(0) \ ::= \ OP\_(1) \ [\ => \ OP\_(0) \ ] \\
OP\_(1) \ ::= \ OP\_(2) \ \{ (\ `\wedge' \ | \ `\vee' \ ) \ OP\_(2) \ \} \\
OP\_(2) ::= OP\_(3) \ \{ \ (\ `<' \ | \ `\leq' \ | \ `=' \ | \ `\neq' \ | \ `\geq' \ | \ `>' \ ) \ OP\_(3) \ \} \\
OP\_(3) ::= OP\_(4) \ \{ \ (\ `+' \ | \ `-' \ ) \ OP\_(4) \ \} \\
OP\_(4) ::= OP\_(5) \ \{ (\ `*' \ | \ `/' \ | \ `\%' \ ) \ OP\_(5) \ \} \\
OP(5) ::= PRIMITIVE \\
PRIMITIVE ::= \left(\begin{array}{ll}
`(' EXP `)' & | \\
STATEVAR & | \\
VAR & | \\
CONST \ [\ `(' EXP \ \{ \ `,' \ EXP \ \} `)' \ ] &
\end{array} \right)
\end{array}
$$

Table 3.1: Abstract Syntax of Expressions

3.0.0.4 Terminals

The remaining elements of the grammar come directly from the AST.

$$
\begin{array}{l}
STATEVAR ::= \ svar \ PLAINNAME \\
VAR ::= \ var \ \ NAME \\
CONST ::= [\ const\ ]\ \ (\ ?\ alpha\ ?\ |\ ?\ greek\ ?\ )\ \ \{\ ?\ allcharacters\ ?\ -\ \text{`}\backslash\text{'}\} \\
NAME ::= \ PLAINNAME\ \{\ \text{"}/\text{"}\ |\ \text{"}\cdot\text{"}\ \} \\
PLAINNAME ::= \ \ (\ ?\ alpha\ ?\ |\ ?\ greek\ ?\ )\ \ \{\ ?\ alpha\ ?\ |\ ?\ greek\ ?\ \ |\ 0-9\ |\ \text{`}\_\text{'}\}
\end{array}
$$

Table 3.2: Abstract Syntax of Terminals

To reduce the need for redundant terms in the examples, the explicit use of the keyword "*const*" is optional since *CONST* is processed last after all other production rules have failed. The backslash character is reserved for future use and cannot be used in the concrete syntax.

3.1 Synthesis and Analysis

The concrete syntax described in the previous section presents a form in which meaningful input strings can be written and converted by a parser into an AST. The concrete syntax allows us to present examples of the components that make up a document.

In this section, common input strings are synthesized for the purpose of analyzing their meaning and semantics. Examples are provided in this section, along with a description of the role and significance of the component in forming a query.

3.1.0 Constants and User-defined Functions

Constants are used for making assertions that hold for the entirety of the document, not just one step. Two distinct types of constants are recognized. Those which consist entirely of digits prefixed with optional sign and followed optionally by a decimal point and any number of zeros are interpreted as integers. Otherwise the constant must be a built-in keyword, or a user-defined symbol. User-defined symbols start with an alphabetic or greek character and contain any combination of greek and alphanumeric characters and underscores.

Type information for the constant will be inferred by the typer. For example, in the expression *const* 0 + gcd(*const* 231, *const* 65) the strings gcd and + will be parsed into the ASTNodes Const("gcd") and Const("+") respectively, both of type FunType(IntType, FunType(IntType, IntType)).

3.1.1 Application

The AST for SIMPPLE supports application to one argument at a time, so during parsing, applications are rewritten in curried form. For example, the expression *const* 0 + gcd(*const* 231, *const* 65) is transformed during parsing into the AST Apply(Apply(Const("+"), Const("0")), Apply(Apply(Const("gcd"), Const("231")), Const("65"))).

One caveat with regard to functional application is that further restrictions may be placed on their use depending on the oracles available. Specifically, it is safe to assume that all oracles support the fully-applied use of any function. The same cannot be said for partial application. As an example, the expression gcd 42 60 is expressible

in all target oracles, but not the partially applied expression $\gcd 42$.

3.1.2 Specification

When logical expressions are enclosed in angle brackets, they have a special syntax. First and foremost, the angle brackets denote that the logical expression is to be treated as a specification; that is, a Boolean function of two states. The angle brackets allow us to expose the state variables of interest within the two state arguments. Before presenting an example of a specification, it is important to understand ASTNames and the annotations they support.

3.1.2.0 ASTNames

ASTNames are used within a specification to refer to variables. Undecorated names may be used in general; ASTNames decorated with Prime or Dot are reserved for state variables.

3.1.2.1 Annotations on Names

Within angle brackets are three distinguished classes of decorated names for state variables: plain, primed and dotted. The convention is that a plain decoration refers to the values mapped by the source state. Variables decorated with Primed names refer to values mapped by the target state. Existentially or universally bound intermediate state variable may also occur within angle brackets. Names decorated with Dotted notation are reserved for this purpose. A specification that swaps the values held by two state variables a and b can be expressed in a SIMPPLE document as:

$$\langle svar\ a' = svar\ b \wedge svar\ b' = svar\ a \rangle$$

The specification above indicates that the values held in state variables a and b must be swapped one for the other upon termination of the program. The angle brackets expose the state variables of interest within the two states. As mentioned in the previous paragraph, plain variables within angle brackets refer to variable names mapped to values by the initial state s. Primed variables within angle brackets refer to variable names mapped to values by the final state s'.

Currying converts the application of functions to arguments one argument at a time. Taking this into account, the final AST for the expression is Spec(Apply(Apply(Const("∧"), Apply(Apply(Const("="), StateVar(Prime("a"))), StateVar(Plain("b")))), Apply(Apply(Const("="), StateVar(Prime("b"))), StateVar(Plain("a")))).

3.1.3 Computer Instructions

A computer instruction is a specification for which a compilable algorithm has been provided. In SIMPPLE, these instructions exist only outside of angle brackets. Two important computer instructions are built into the grammar: state-variable assignment, and state-variable declaration.

3.1.3.0 Assignment

The value of a state variable is modified by assigning it a new value computed from the input state. In the example the present value of b is added to the present value

of a to become the new value for subsequent states in the path of execution. In the concrete grammar presented in this chapter, this would be written:

$$assign\ svar\ a := svar\ b + svar\ a$$

The corresponding SIMPPLE AST is given by Assign(StateVar(Plain("a")), Apply(Apply(Const("+"), StateVar(Plain("b")))), StateVar(Plain("a"))))

3.1.3.1 Declarations

Typically some state variables are present in the context by virtue of the established path of execution. The grammar supports the declaration of new state variables as needed. A state variable is introduced by assigning an initial value to it along with a block within whose scope the state-variable exists. A typical example would be to introduce a temporary variable to swap the values of two other variables. In the concrete grammar presented in this chapter, this would be written:

$$declare\ svar\ t := svar\ a\ in\ \ (assign\ a := svar\ b;\ \ assign\ b := svar\ t)$$

The corresponding SIMPPLE AST representation as shown in Figure 3.1. With the possible exception of the semicolon, the transformation of the input string is straightforward. For added clarity, the original input string and the output AST is placed side-by-side to highlight the correspondence between the concrete and abstract form.

When comparing the concrete and abstract syntax, the treatment of the semicolon

```
DECLARE
        t :=
             svar a
in (
      assign  svar a := svar b ;
      assign  svar b := svar t

)
```

```
StateVarDecl(
                Plain("t"),
                StateVar("a"),
                Apply(          Apply(  Const(";"),
                                        Assign(StateVar("a"),StateVar("b"))
                                ),      Assign(StateVar("b"),StateVar("t"))
                )
)
```

Figure 3.1: General Swap AST

is slightly more involved and requires some explanation. In the concrete representation, the semicolon is unobtrusively written in a readable repeated infix notation. In the abstract representation, the semicolon is treated as an operator, and is applied to its arguments one at a time using the technique of Currying. Scala's parser combinators easily support the conversion from the infix notation to the curried version of application supported by the AST.

3.1.4 User-defined Functions

The introduction of functions and constants beyond those defined by the system is of utmost importance. For instance, the user may wish to define functions such as the greatest common divisor `gcd`. By making assertions about the values of functions under application to constant or variable expressions, it is possible to define the behaviour of these functions.

From the point of view of the parser, user-defined symbols are all simply constant strings of text, and provided they do not begin with an system-defined escape character and do not conflict with a system-defined symbol, the symbols are treated as free expressions of the language. Type information for the user-defined functions will

be inferred by the typer.

3.1.4.0 Quantified Lambda Abstractions

User-defined functions are constrained by quantified lambda abstractions. The quantifier may be existential (exists) or universal (forall). Quantifiers are converted internally to constants of the same name and applied to an implied, lambda abstraction as its sole argument. The concrete syntax described in this chapter does not provide any method for expressing lambda abstractions.

Lambda abstractions are associated with a right-side AST and a bound ASTName. The name will be registered as a variable and is therefore subject to the same rules as leaves of type Var. The lambda abstraction is a function that accepts a value or expression which is to be substituted for every bound occurrence of the ASTName within the body. When a quantifier is applied, the result is a Boolean expression.

As an example, the GCD algorithm has the following property:

$\forall i \cdot \ var\ i > 0 \ \Rightarrow \gcd(var\ i,\ 0) = var\ i$

If we represent the parsed tree for the body $var\ i > 0 \ \Rightarrow \gcd(var\ i,\ 0) = var\ i$ using the notation: $\mid var\ i > 0 \ \Rightarrow \gcd(var\ i,\ 0) = var\ i \mid$, then in response to this input string, the parser generates the AST:

$\mid \forall i \cdot \ var\ i > 0 \ \Rightarrow \gcd(var\ i,\ 0) = var\ i \mid$ = Apply(Const("forall"), Lambda(Var(Plain("i")), $\mid var\ i > 0 \ \Rightarrow \gcd(var\ i,\ 0) = var\ i \mid$))

The AST has type: BoolType().

3.2 Summary

This section provides the BNF grammar for the concrete syntax used for formulating server-side queries and shows how it is mapped to the abstract syntax presented in Section 2.2. In Chapter 4, the reader will be able to recognize and write test cases developed for the server-side software.

Chapter 4

Client/Server Interface

The language discussed in this document allows the client software to formulate a query asking the server whether one command is refined by the other. In the previous chapter, a concrete syntax for the client/server interface was presented. The client-side user-interface converts changes in a document to a server-side query. The code developed in this thesis implements the server-side logic. Implementation of the client-side user-interface is left to a future phase of development. In order to test the server-side, a mock-up is used in this section that allows presentation of the inputs the client-side user-interface must accept and the server-side query it may output in response.

4.0 Client-side Input/Output Requirements

The editing environment requires a number of elements, as illustrated by the mock-up in Figure 4.0. The input to the client-side user-interface includes a previous revision of the document to serve as a previous step in the refinement process. The active

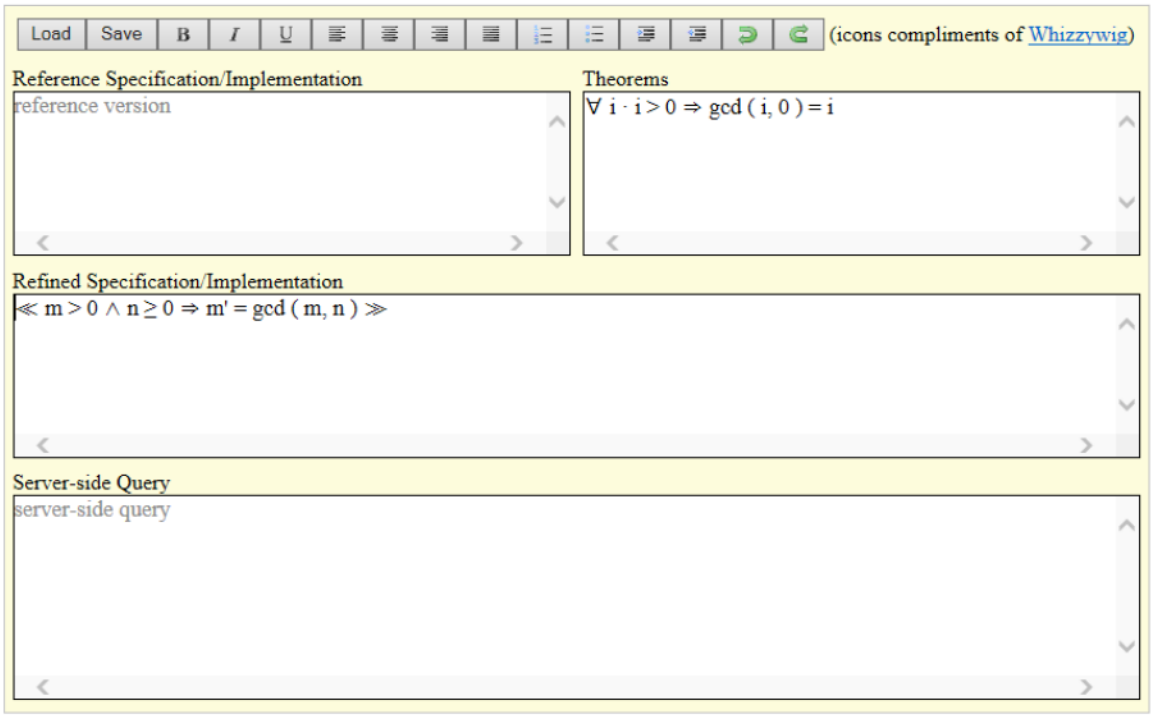


Figure 4.0: Specification of GCD in Editor Mockup

document forms the current step. Theorems justifying the revision can be entered and maintained separately by the user.

The theorem presented in Figure 4.0 is the trivial solution of the GCD algorithm, which is a subset of the overall solution. The case introduction law provides the rules by which to implement a specific case in a way that does not limit the overall specification of the problem.

SIMPPLE allows rules to be applied using both conventional and fine-grained formal methods. Automated verification of proofs is provided by the server-side oracles. As the user edits the document, the editor formulates a query to pose to the server-side prover to check the correctness of the user's modifications. The output required of the client-side user-interface is the server-side query. A read-only window

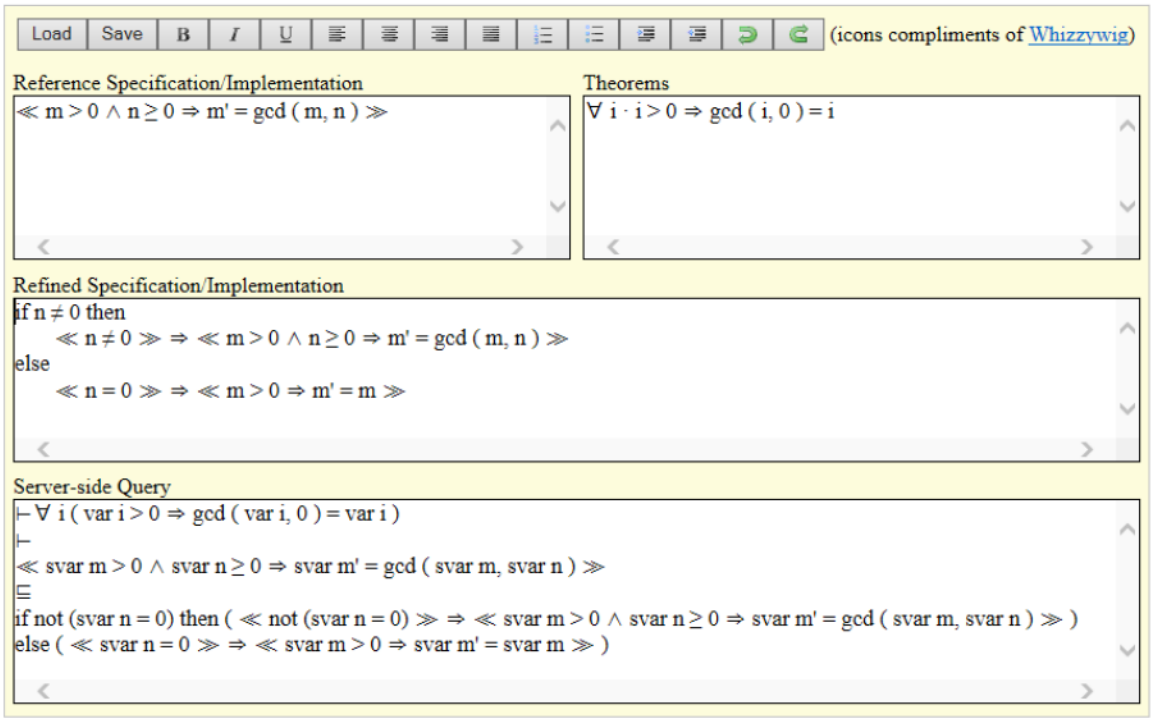


Figure 4.1: Trivial Case in Editor Mockup

displays the server-side query for diagnostics and development purposes.

In the example shown in Figure 4.1 the user has applied rules that allow them to introduce the case where the second parameter to the GCD function is 0. In order to verify that the original specification is not violated by the proposed implementation, a query asking whether the previous step in the document's history is refined by the current step is formulated in the concrete syntax of the server. In the absence of a client-side user-interface, such queries can also be manually entered and sent directly to the server.

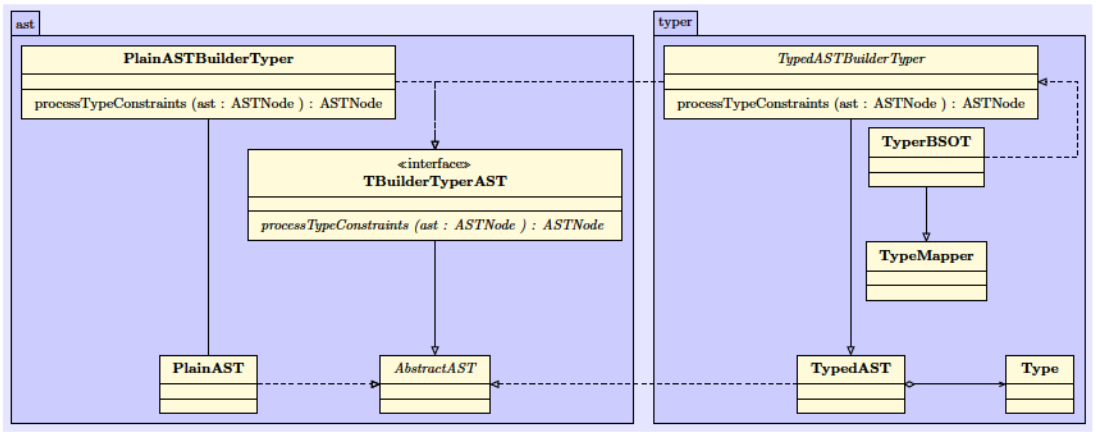


Figure 4.2: Builder/Typer Class Diagram

4.1 Server-side Implementation Details

This section describes details of the functions that are required to parse input strings and store them as TypedASTs. The relationship between the classes required to implement the full builder interface is depicted in Figure 4.2.

4.1.0 Builder For Typed ASTs

Construction and typing of input strings during parsing is facilitated by a polymorphic trait TBuilderTyperAST. It declares an abstract function processTypeConstraints (ast : ASTNode) : ASTNode to serve as the primary interface between it and concrete builder typers. Concrete builder typers are expected to apply type constraints to the ASTNodes as the tree is built.

Figure 4.3 shows the class methods of the TBuilderTyperAST class. Each of the static builder methods combines construction and typing of ASTNodes by calling the appropriate constructor and then passing the constructed object to the abstract function: processTypeConstraints (ast : ASTNode) : ASTNode. The processType-

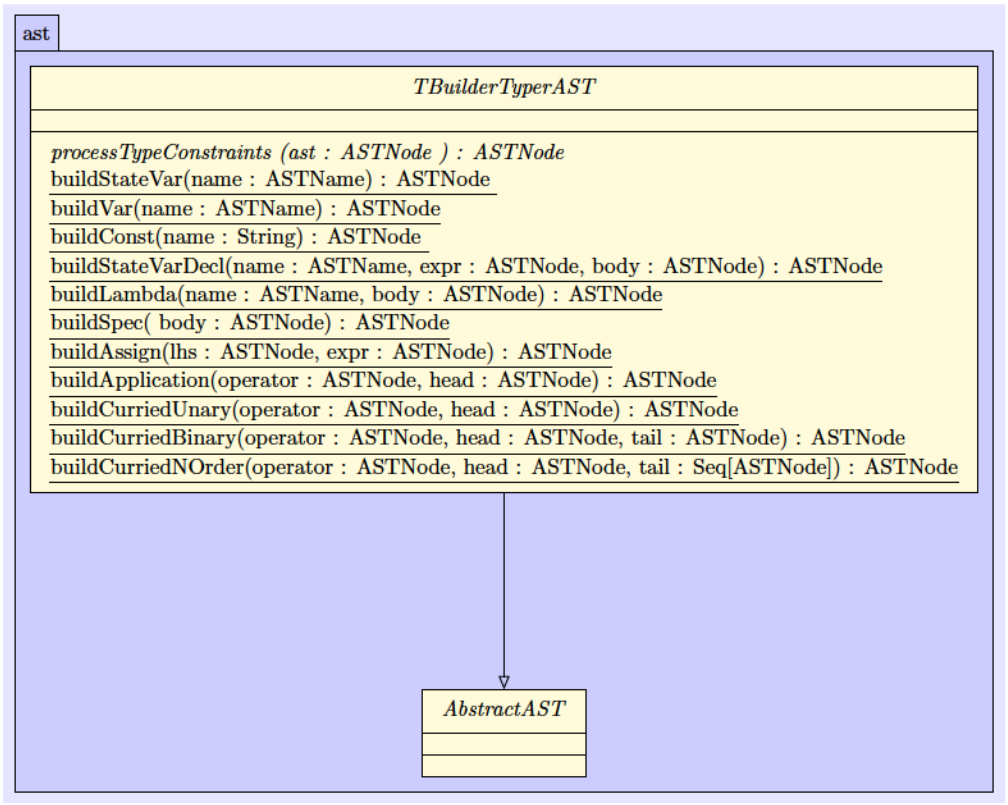


Figure 4.3: Builder/Typer Class Methods

Constraints function accepts an untyped AST whose children have partial or no type constraints. The default implementation returns an identical AST whose type has been set to a default value of initTypeInfo. Any subclass of AbstractAST can define its own type system for the generated AST. By overriding processTypeConstraints, subclasses can override the default implementation to return an AST whose type has been inferred as much as possible given the set of type constraints.

4.1.0.0 PlainASTBuilderTyper

A corresponding concrete builder/typer is derived from the TBuilderTyperAST: it is named PlainASTBuilderTyper. The primary task of the concrete builder is to provide

the polymorphic abstract builder with a reference object (in the case of PlainASTBuilderTyper, the reference is the companion object of PlainAST). The reference object serves as a central interface by which the TBuilderTyperAST can access public functions of the typed AST. An equally important function of PlainASTBuilderTyper is to override the default implementation of processTypeConstraints.

4.1.0.1 TypedASTBuilderTyper

As with PlainASTs, a concrete builder derived from the TBuilderTyperAST is required in order to provide an implementation of processTypeConstraints for TypedASTs. The implementation requires a mechanism for managing changes to the state space and for keeping track of the types of variables and constants that are not part of the state space. A fully developed typer exists in Haskell for this purpose but is not in use in the current Scala implementation. Abstract methods are currently declared until such time as the typer is available. The following functions are required of the interface as declared by TypedASTBuilderTyper:

- queryStateSpace() : Type;
- declareStateVar(id : ASTName, ty : Type) : Type;
- undeclareStateVar(id : ASTName) : Type;
- queryBoundType(name : ASTName, expr : ASTNode) : Type;
- queryIdType(id : String) : Type;
- querySymbolType(symbol : String) : Type;

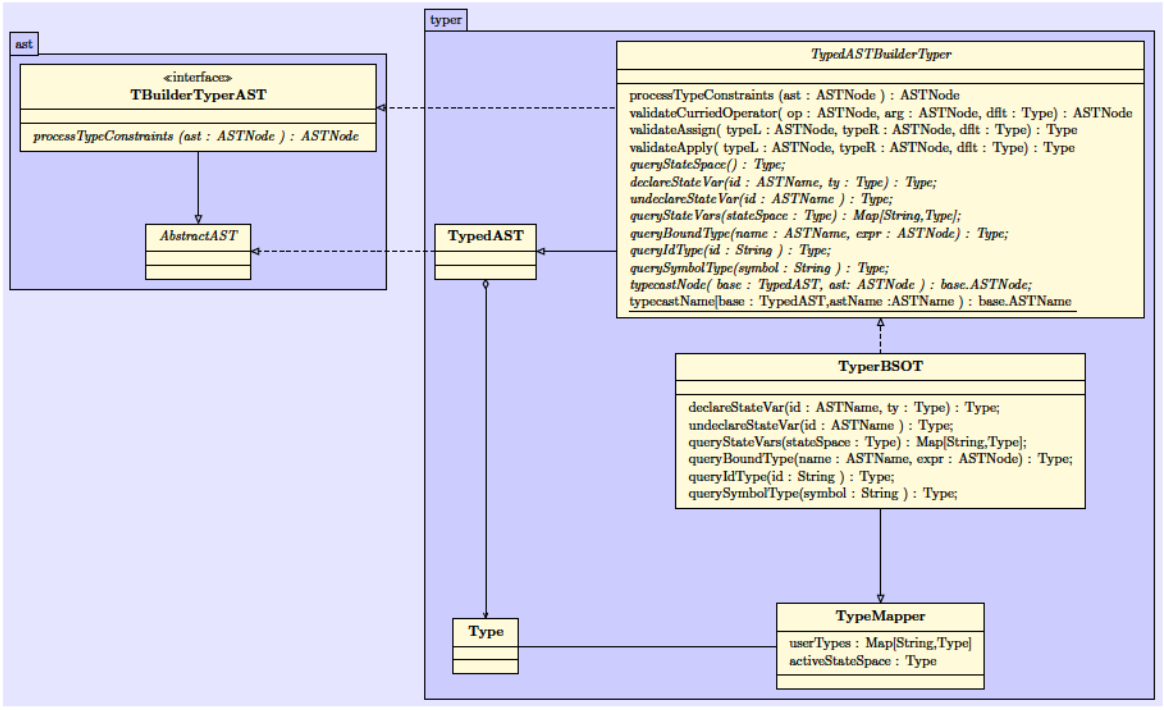


Figure 4.4: Classes Used for Typed ASTs

4.1.0.2 TyperBSOT and TypeMapper

The declaration of an abstract interface as part of a TypedASTBuilderTyper means it has no direct implementation. TyperBSOT provides a basic implementation of the interface defined by TypedASTBuilderTyper. It suffices for the purposes of the thesis to use a declarative TypeMapper to assign types by mapping their names to user-declared types. TyperBSOT mixes the TypeMapper into the TypedASTBuilder-Typer to inherit this behaviour. Also inherited from TypeMapper is a stack-based implementation for allocating state spaces: each new state variable declared pushes a distinct, unique state space constant onto the stack where it remains as long as the body of each state variable declaration is in scope. Figure 4.4 provides a high-level description of the classes and relationships involved.

4.2 Summary

In the first half of this chapter, an example of a server-side query was presented in the concrete syntax of SIMPPLE. A mockup of the client-side user interface was used for presentation purposes. It must be understood by the reader that until such time as the client-side user-interface is written, testing of the server-side interface is achieved through unit testing (e.g. ScalaTest).

The second half of this chapter is devoted to introducing the server-side code developed under this thesis. The introduction starts with a description of the routines that build and type individual ASTNodes. These routines are used in conjunction with Scala's built-in parser combinators to convert test-strings into complete ASTs.

Chapter 5

Converting ASTs to FOL

This chapter presents a summary of the conversion of ASTs presented in the previous chapter into a first-order representation. Details of an implementation plan for higher-order logic are given in Chapter 8.

5.0 Reduced AST for First-order Subset

ASTs can be translated to a first order equivalent. The first order translation is a distinguished subset of the overall language of SIMPPLE. The following nodes are not present in the first order representation:

$Spec(expr : ASTNode) : ASTNode$

$StateVarDecl(n: ASTName, expr : ASTNode, body : ASTNode) : ASTNode$

$Assign(lhs : ASTNode, expr : ASTNode) : ASTNode$

After translation, the original tree is expressed in terms of the case classes listed in Table 5.0.

Additionally, applications of higher-order functions such as the "if-then-else" ternary

operator, guarded blocks, and sequential compositions are converted to a first-order form. The specifics of this conversion are discussed in Section 5.2.1.

$Apply(f\text{: }ASTNode, expr : ASTNode) \ : ASTNode$
$Lambda(n\text{: }ASTName, expr : ASTNode) \ : ASTNode$
$StateVar(n\text{: }ASTName) \ : ASTNode$
$Var(n\text{: }Name) \ : ASTNode$
$Primed(n\text{: }ASTName) \ : ASTName$
$Dot(n\text{: }ASTName) \ : ASTName$
$Plain(s\text{: }String) \ : ASTName$
$Const(s\text{: }String) \ : ASTNode$

Table 5.0: Case Classes of Reduced, First-order ASTs

5.1 AST Normal Form

The ASTNormalForm trait is the central player in the Scala source code. It brings together the high-level definitions needed to build an expressive language with the low-level builders needed to remove and translate these high-level expressions into a common first-order language equivalent. The ASTNormalForm is combined with parsers to impart these capabilities to classes that can accept strings as inputs. The resulting configuration of classes is shown in Figure 5.0

The translation of ASTNodes to first-order is handled in Scala using a match-case construct to recognize trees that are excluded from the language or which require special handling. In order to support a modular methodology to building and extending the built-in operations supported by the proof-system, traits associated with ASTNormalForm can be mixed in as the program matures using the multiple inheritance feature of Scala. Each mix-in trait has the power to declare further high-order

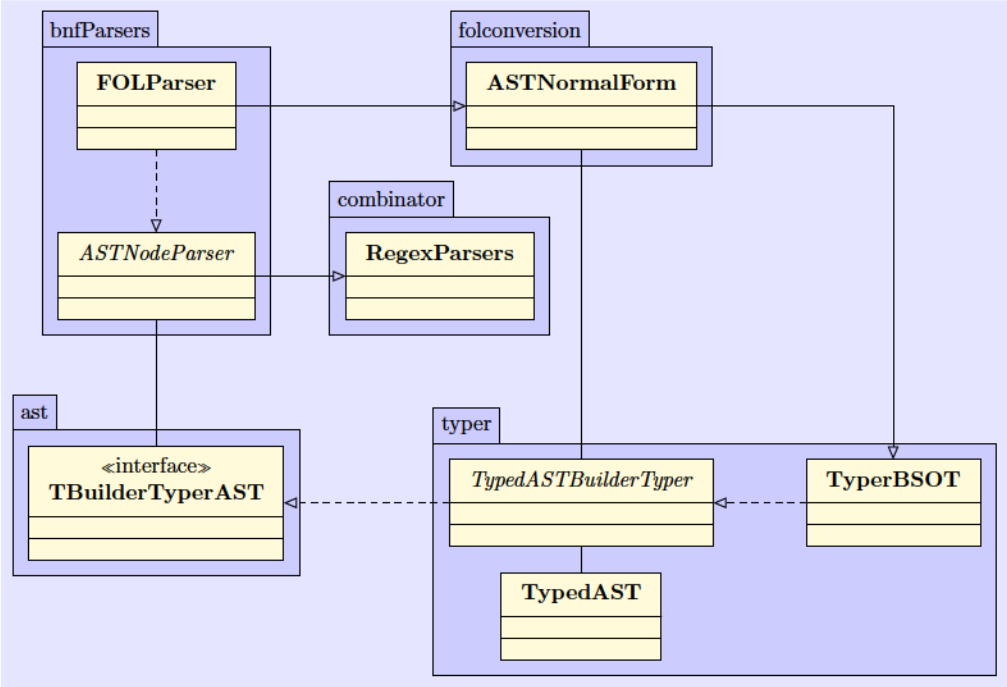


Figure 5.0: ASTNormalForm Class Diagram

constants. Traits in support of refinement, integer arithmetic, and common programming blocks are included in the existing source and are used to fully parse the examples presented in the thesis. The relationship between these classes is depicted in Figure 5.1

5.1.0 Programming Theory

Nodes of type $Type.CommType(\alpha : \Sigma)$ are removed in the translated form and replaced by Boolean-valued expressions. To remove these nodes, an equivalent expression must be formed using the normalized first-order subset of the language. The axioms, theorems and definitions of programming theory used in this thesis are adapted from ToC.[Norvell, 2012]

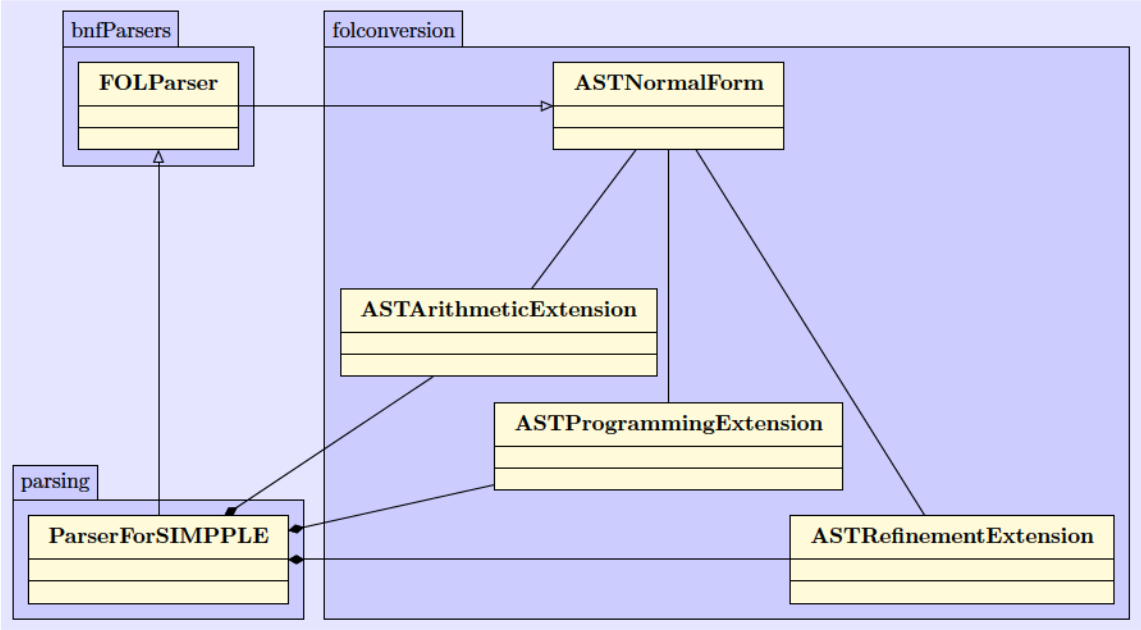


Figure 5.1: ASTNormalForm Parser Extensions

5.1.0.0 State Variables

State variables are preserved during translation. Notationally this can be expressed as:

$$\| \mathit{StateVar}\,(\,v\,)\| \;=\; \mathit{StateVar}\,(\,v\,)$$

5.1.0.1 Constants

Constants representing integers, Booleans, and associated operators, as well as constants with no reserved meaning are preserved without modification in the translated AST. The specifics of the conversion of other constants with a reserved interpretation are discussed in section 5.2.1.

5.1.0.2 Spec

The concrete syntax permits only those ASTNodes that are allowed in the reduced AST to be parsed into Spec ASTNodes. The translation of Spec is therefore straightforward: the body is translated and replaces the original spec. Notationally this can be expressed as:

$$\| Spec\,(body) \| = \| \; body \|$$

Specs are ASTNodes of type CommType($\alpha : \Sigma$). Σ is used to determine Vars that correspond to state variables in the translated AST. After the translation, the resulting tree will have the type BoolType(). Details of this translation are provided in Section 5.3.

5.1.0.3 Quantified Lambda Abstractions

A quantified lambda abstraction has the form Apply(Const(quantifier),Lambda(v,body)) where the quantifier is $\exists$ ("exists") or $\forall$ ("forall"). The translation is required only to ensure that the body is in first order form. This is described as:

$$\| \; Apply\,(\; Const\,(\; quantifier\;)\,, \lambda\;(\; v,\; body\;))\;\| \;=$$
$$Apply\,(\; Const\,(\; quantifier\;)\,,\;\; \lambda\;(\; v, \| \; body\; \|))$$

5.1.0.4 Assignment

The assignment command is represented in the AST by the case class Assign(target, source) where both target and source are ASTNodes. This representation is quite expressive supporting, among other possibilities, systems of linear equations. In this thesis, only assignment to a single StateVar is considered. When the target of an

assignment is a single state variable, the only state variable affected by the instruction is the target. All other state variables must be unaltered.

To translate the Assign ASTNode, the left-hand side is searched for a plain state variable. When a plain state variable is found, it is translated into a primed variable of the same name. The primed variable is asserted equal to the expression formed by translating the source or right-hand side. The resulting equality is combined using the conjunctive AND operator with other assertions that ensure no side-effects result from the assigment. These other assertions are built by performing a finite quantification for all variables defined in the state space, so that every other variable declared in the state space is asserted to have a final value matching its initial value.

Considering only the case when the target is a single StateVar, this can be written as:

$$
\begin{aligned}
&\| \mathit{Assign}(\ \mathit{StateVar}\ (\ \mathit{Plain}(\ v\ ))\ ,\ \mathit{rhs}\ )\ \|_{\Sigma} = \\
&\quad \mathit{Apply}\,(\mathit{Apply}\,(\mathit{Const}\ (=)\,, \mathit{StateVar}\,(\mathit{Prime}(\text{`}v\text{'}))\,, \|\mathit{rhs}\|_{\Sigma}) \\
\bigwedge\ &\quad \mathit{Apply}\,(\mathit{Apply}\,(\mathit{Const}\ (=)\,, \mathit{StateVar}\,(\mathit{Prime}(\text{`}w\text{'}))\,, \mathit{StateVar}\,(\mathit{Plain}(\text{`}w\text{'}))) \\
\bigwedge\ &\quad \mathit{Apply}\,(\mathit{Apply}\,(\mathit{Const}\ (=)\,, \mathit{StateVar}\,(\mathit{Prime}(\text{`}x\text{'}))\,, \mathit{StateVar}\,(\mathit{Plain}(\text{`}x\text{'}))) \\
\cdots\ &\quad \cdots \text{ repeat for additional state variables in } \Sigma
\end{aligned}
$$

5.1.0.5 StateVar Declaration

StateVarDecl is unique in that it modifies the value of the state-space α of type Σ over the scope of its body. The conversion depends on two functions defined as part of TypedASTBuilderTyper (from which ASTNormalForm is defined), namely declareStatevar and undeclareStatevar. First, the type of the expression and the variable is inferred and the variable is added to the state variable namespace by

calling declareStatevar. A tree representing the sequential composition of the Assign tree and the body is then made and translated. The state variable is removed from the state variable namespace by calling undeclareStatevar, and its final, primed instance is existentially bound to the translated AST. This is seen more simply in the notation:

$$\| \ StateVarDecl\ (\ v\ ,\ initExpr : ExprType(\alpha, T)\ ,\ body\ )\ \|_{\Sigma} \ =$$
$$Apply\left(\ Const(\ \exists\ ), \lambda\left(\ Prime(\ v\ ),\ \| \ v\ :=\ initExpr\ ;\ body\ \|_{\Sigma \cup (v,T)}\ \right)\ \right)$$

5.2 Mix-in Extensions

The ASTNormalForm trait is the central player in the Scala source code. It defines the algorithm by which ASTs can be built and subsequently translated to first order. Certain core requirements of the ASTNormalForm class are dictated by the AbstractAST. Other implementation details depend on the concrete syntax and the theories used to convert higher-order semantics into a first-order form.

In order to support changes in concrete syntax from the core features, the class supports mix-in functions. These can be combined with the ASTNormalForm to extend the default implementations for the following methods:

- translate(ast : ASTNode) : ASTNode

  Call this function to translate an AST to first order.

- translateStateSpaceModel(translation : ASTNode) : ASTNode

  Override this function to implement other models (including high-order models) for state spaces.

- translateStateSpaceModelTypes(ty : Type) : Type

  Override this function to implement other type-systems (including high-order type-systems) for state spaces.

- def validateCurriedOperator(operator : ASTNode, argument : ASTNode, failedResult : Type) : ASTNode

  Override this function to assign types to dependently typed operators introduced by a mix-in extension.

- typecastNode(base : TypedAST, ast : ASTNode) : base.ASTNode

  Once the translation is complete, the ASTNormalForm can be discarded and the tree can be passed to any other class derived from TypedAST. This function performs the necessary type casts to demote the AST to one compatible with the base type.

Traits in support of refinement, integer arithmetic, and common programming blocks are included in the existing source and are used to fully parse the examples presented in the thesis. Details of the class methods implemented by each of the traits is shown in Figure 5.2. Specific details on each of the methods and how they participate in the conversion to first order are given later in this section.

5.2.0 Refinement

At the highest level, a document in SIMPPLE is a query. Each query is a collection of trees. The concrete syntax for a query is given as:

$$QUERY ::= \{ \vdash \ TREE \ \{ \sqsubseteq \ TREE \} \}$$

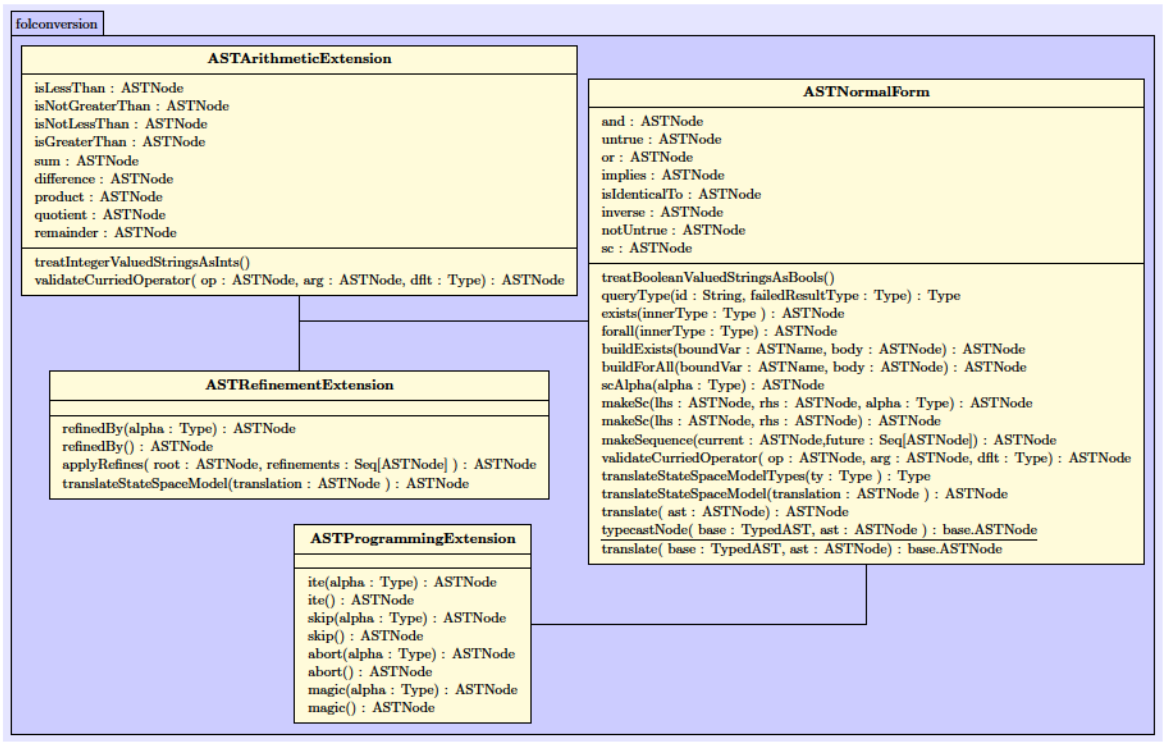


Figure 5.2: Mix-in Extensions

Theorems and derivations are not a formal part of the AbstractAST, and are creatures of the parser. The parser starts at the beginning of the document and searches for the first theorem or derivation. It parses the series of trees, and returns all the trees in the document up until the start of the next theorem or derivation. If only one tree is returned, the tree is a theorem. Otherwise, the sequence of trees is a derivation consisting of continuing refinements.

Formally, any *TREE* is typed by the typer as a function of two states. Refinement, being a preorder between *TREEs* is thus a higher-order operation that operates on functions that are not fully applied. This interpretation can be deferred in most cases until after the AST is ready for translation to first-order. ASTRefinementExtension class implements the methods for deferring this interpretation.

5.2.0.0 ASTRefinementExtension

ASTRefinementExtension defers the interpretation of refinement as a first-order function by declaring a higher-order operator named refinedBy(). This high-order function is added at the parsing stage.

During parsing, if a theorem without proof is encountered, it is added to a sequence of theorems maintained by the parser. When a derivation is parsed instead, the applyRefines method of the ASTRefinementExtension class builds the AST from the head and tail of the derivation, building the case class constructor: Apply(Apply(refinedBy(),head),tail.head). It adds the resulting tree to a sequence of derivations, and then recurses until ASTs for each refinement have been added to the list of derivations.

The parser continues parsing the remainder of the document in a similar way. The result is two sequences

$Theorems\,(s : String) : Seq\,[ASTNode]$

$Derivations\,(s : String) : Seq\,[ASTNode]$

When it is time to translate the AST to first order, the ASTRefinementExtension matches the AST against the extractor for the case class Apply(Apply(refinedBy(), head), tail.head). If a match is detected, ASTRefinementExtension provides the following replacement:

$$\| \; Apply\,(\; Apply\,(\; refinedBy(),\; tree\;),\; refinement\;)\; \|_{\Sigma} \;=$$

$$Apply\,(Const(\forall), \lambda\,(Prime(\text{`}v\text{'}),\; Apply\,(Const(\forall), \lambda\,(Plain(\text{`}v\text{'}),$$

$$Apply\,(Const(\forall), \lambda\,(Prime(\text{`}w\text{'}),\; Apply\,(Const(\forall), \lambda\,(Plain(\text{`}w\text{'}),$$

$\cdots$ repeat for additional state variables in Σ

$$Apply\,(Apply\,(\quad Const\,(\Rightarrow),$$

$$\|refinement\|_{\Sigma}),$$

$$\|tree\|_{\Sigma}) \qquad\qquad \cdots))))$$

5.2.1 Application (First-order and High-order)

When a high-order application of a constant is encountered, the mix-in trait which defines it must provide a suitable first-order translation (either during parsing or in the final translation to first order). Any constants defined to have special interpretation by the extension should be intercepted by the extending trait and should not be sent to the super class for first-order processing. This process often happens in the context of substituting an application of the constant to one or more of its arguments. In these cases, the application of the constant is typically translated in tandem with its arguments.

Besides refinement, there are a number of higher-order functions defined in the concrete syntax that need to be intercepted and given special treatment during conversion to first-order. Some of these constants, like the Sequential Composition operator are defined in the ASTNormalForm companion object. Others, including the refinement operator and "if-then-else" blocks are defined in companion objects belonging to various extensions (mix-in traits).

Besides ASTRefinementExtension, two other examples of mix-in extensions to the

ASTNormalForm exist in the implementation.

- ASTArithmetricExtension

  Declares typed-operators for curried arithmetic operations typically associated with integers. Since many of these operations can also operate on other types (i.e. reals, complex, and matrixes), the operators are typed when built based on a type-schema. For instance, relational operators acting on an argument of type ty are assigned a type ty->ty->BoolType().

  Also provided is an option for enabling automatic detection and typing of integer strings as IntType().

- ASTProgrammingExtension

  Declares typed operators representing the programming specifications magic, skip and abort, as well as the ternary command "if-then-else".

5.2.1.0 Guarded Expressions

The concrete syntax supports the following high-order operation:

$TREE ::= BLOCK\ [\ \Rightarrow BLOCK\ ]$

This high-order operator has the type FunType(CommType(α), FunType(CommType(α), CommType(α))). As described in Section 5.3, after translation, each *BLOCK* is type-converted to BoolType(), and the operator "⇒" becomes normal implication with the type FunType(BoolType(), FunType(BoolType(), BoolType())).

5.2.1.1 Sequential Composition

The concrete syntax supports the following high-order operation:

$$BLOCK ::= STATEMENT \quad \{ \quad ; \{ ; \} \; [STATEMENT\ ] \; \}$$

When statements are separated by one or more semicolons, the semicolon is interpreted as the high-order sequential composition operator. The translation of this command is done in two phases. First, the inner commands are translated to first-order as in:

$$\| \; Apply\,(\; Apply\,(\; Const\,(\;;\;)\,,\; lhs\;)\,,\; rhs\;) \; \| =$$
$$Apply\,(\; Apply\,(\; Const\,(\;;\;)\,, \| \; lhs \; \|)\,, \| \; rhs \; \|)$$

Second, the translation can be overridden to support different state space models. For the default state space model, the following steps are used to transform the application to a first-order representation:

0. An extra dot is appended to existing dotted variables in the left-hand term; this includes any dotted variables bound by existing lambda abstractions. Primed state-variables are then replaced with single-dotted variables. A function called scLHS is used to perform this translation.

1. An extra dot is appended to existing dotted variables in the right-hand term; this includes any dotted variables bound by existing lambda abstractions. A function called scRHS is used to perform this operation.

2. The conjunction of the translated left and right hand expressions is formed. The single-dotted variables in the resulting conjunction are bound using an existentially quantified lambda abstraction.

The overall result of the translation is that plain state-variables on the right-hand side and primed state-variables on the left-hand side are replaced with single-dotted variables and the dotted variables are existentially bound. This creates the set $\{var^{\cdot}\}$ which represents an intermediate state $s^{\cdot}$, forming a path that joins the target state of the first *STATEMENT* to the source state of the second. This can be seen more simply in the notation:

$$\| \mathit{Apply}\,(\ \mathit{Apply}\,(\ \mathit{Const}\,(\ ;\ )\,, \| \mathit{lhs} \|)\,, \| \mathit{rhs} \|)\ \|_{\Sigma} =$$

$$\mathit{Apply}\,(\mathit{Const}\,(\exists)\,, \lambda\,(\mathit{Dot}(\text{`}v\text{'}),$$

$$\mathit{Apply}\,(\mathit{Const}\,(\exists)\,, \lambda\,(\mathit{Dot}(\text{`}w\text{'}),$$

$\cdots$ repeat for additional state variables of Σ

$$\mathit{Apply}\,(\mathit{Apply}\,(\quad \mathit{Const}\,(\wedge)\,,$$

$$\mathit{scLhs}\,(\|\mathit{lhs}\|))\,,$$

$$\mathit{scRhs}\,(\|\mathit{rhs}\|)) \quad \cdots))$$

5.2.1.2 If-Then-Else Expressions

The concrete syntax supports the following high-order operation:

$$\mathit{BRANCHSTATEMENT} ::= \quad \mathit{if}\ \mathit{CMDEXP}\ \mathit{then}\ \mathit{STATEMENT}$$

$$[\ \mathit{else}\ \mathit{STATEMENT}\ ]$$

This string is parsed into a ternary operation in the form:

$\mathit{Apply}\,(\ \mathit{Apply}\,(\ \mathit{Apply}\,(\ \mathit{Const}\,(\ \mathit{ite}\ )\,,\ \mathit{guard}\ )\,,\ \mathit{ifClause}\ )\,,\ \mathit{elseClause}\ )\,.$

If the optional else *STATEMENT* is not provided, then *elseClause* defaults to $\mathit{Const}\,(\ \mathit{skip}\ )$.

The translation of the AST is handled by the default translation for application as defined in section 5.2.1.3.

5.2.1.3 Application Of Function

If not intercepted by an extending trait, application is processed by ASTNormalForm using a default method. All user-defined functions are handled using the default method. Also, many built-in operators are supported through this default procedure, including those for boolean logic, and those residing in ASTArithmetricExtension for common integer operators. The default procedure recursively translates the operator and operand in the following fashion:

$$\| \mathit{Apply}(\ \mathit{operator},\ \mathit{operand}\ ) \| \ = \ \mathit{Apply}(\ \| \mathit{operator} \| \ ,\ \| \mathit{operand} \| \ )$$

5.3 Handling of Typed ASTs

Type information is retained during translation as it contains important information about the state variables in use in the state space, and constraints on the types of functions and variables. As part of translation, type information also needs to be converted to an appropriate first-order type. Most notably, given an expression of CommType(α) or ExprType(α,ty), the translation instantiates a distinct set of global variables $\{var^d\}$ for each state, and substitutes the set of variables for the state. The StateSpaceType alpha determines the set $\{var^d\}$ that corresponds to state variables in the translated AST.

To take into account the fact that initial and final states for the typed AST have been substituted for a distinct set of variables $\{var^d\}$, the following type conversions

are performed:

0. ExprType(alpha,ty)

   The translated type is equal to the translated inner type ty. Notationally this is written:

   $$\| \mathit{ExprType}(\alpha, ty) \| = \| ty \|$$

1. CommType(alpha)

   The translated expression has type BoolType(). Notationally this is written:

   $$\| \mathit{CommType}(\alpha) \| = \mathit{BoolType}()$$

2. FunType(argumentType,resultType)

   The argument and result types are translated, and a new functional type results that returns the translated resultType when applied to an operand whose type matches that of the translated argument type. Notationally, this is simply:

   $$\| \mathit{FunType}(\ t\ ,\ u\ ) \| = \mathit{FunType}(\| t \|, \| u \|)$$

5.4 Summary

This chapter shows how the abstract syntax of SIMPPLE can be translated into first order logic. The default implementation discussed in this chapter provides a first-order interpretation for frequently encountered constants and operators. The chapters to follow show how a document that has been translated in this way is ready to be processed by oracles available to the server. For example, the ASTNodes

translated by the default implementation can be evaluated directly by oracles with support for first-order logic.

Chapter 6

Generic Prover Interface

The thesis has shown how the software uses the ASTNormalForm class to formulate a query and translate it into a first-order translation. At this point, the ASTNormalForm, the parsers and all the mix-in extensions have completed their work. The resulting tree is typecast to a TypedAST.

Constants and symbols defined as part of Boolean, arithmetic and user-defined theories remain in the language. These include the constants representing true, false, conjunction and disjunction, identity and inequality, addition and subtraction, multiplication and division. These constants form the basis for the prover interface through which third-party oracles are accessed.

Figure 6.0 shows the classes involved in the prover interface. In this chapter we show how these classes support adapters to various third-party APIs in order to process the query.

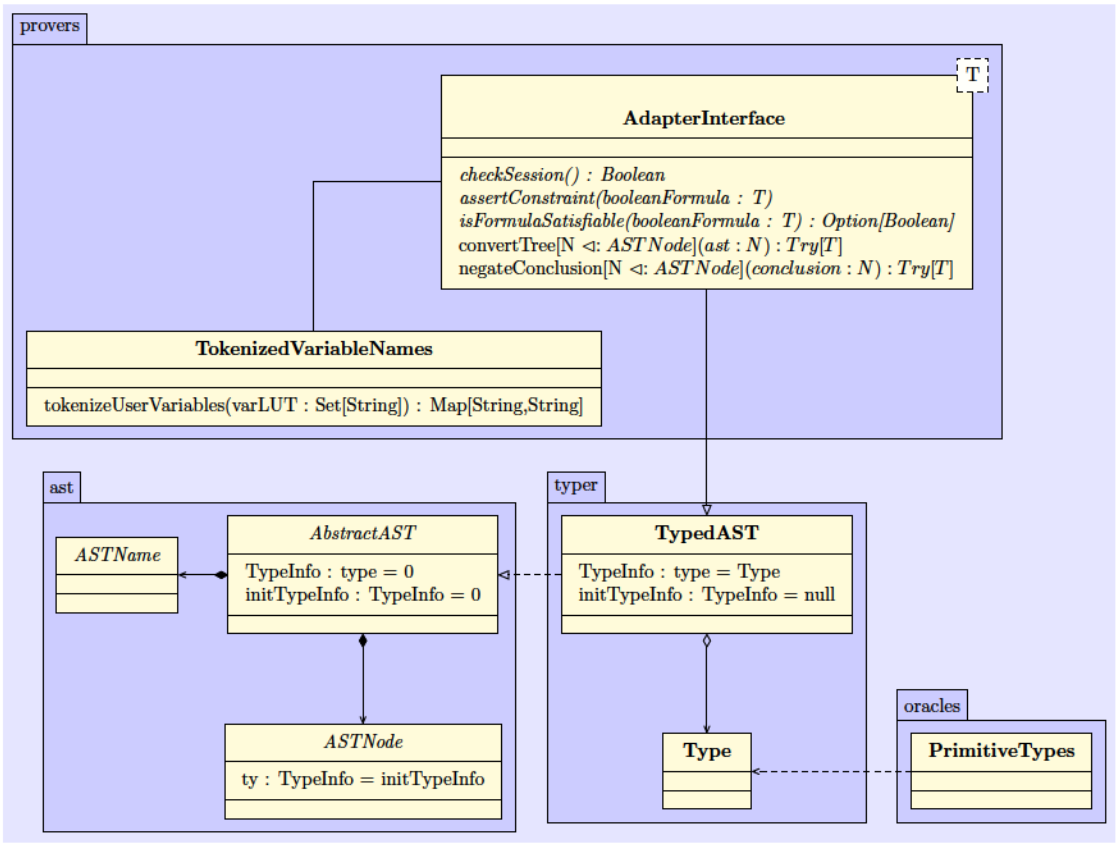


Figure 6.0: Prover Interface Class Diagram

6.0 Oracles

The prover interface prepares the AST for the final step, conversion of the query to the grammar of third-party programs at the server's back-end. Because the generic prover interface establishes an arm's length relationship between the server and these programs, it is helpful to think of them as external oracles. Given an AST, conversion involves one of two options:

0. if the oracle has an API, the AST is processed by translating it and passing it to the oracle using the API.

1. alternatively, the translator can convert the translated AST to the concrete syntax for the oracle (a process similar to pretty-printing).

The arm's length relationship is established by defining a set of constants in a Scala interface which can be adapted for each oracle by optionally mapping them to ASTs native to the oracle. If a pre-defined constant appears in the query but is not in the language of the oracle, then conversion fails for that oracle. Additionally, user-defined constants can be introduced by providing axiomatized definitions to establish the semantics of the constant. Provided that these definitions can be expressed in the language of the oracle, the user-defined constants can be used in expressions converted to the oracles grammar.

6.0.0 Dealing with Types

The examples in this thesis rely upon the TypedAST class. TypedAST is one example of a concrete AbstractAST with a fully developed type system, but it is not the only possible type system. Each type-system augments the AST by providing type information.

Once the typed AST is converted to the native form specific to an oracle, this type information is no longer available. To decouple the oracle from the typed, concrete AST, strings are used by the adapter to report back type information about the converted native form. A set of primitive types shared between the server and its oracles are defined in the class PrimitiveTypes. Currently the list allows for boolean and integer types. Regular expressions are also defined in the class allowing the adapter to test for names of constants that are intended to be associated with

a primitive type. For example, a constant "0" whose name matches the regular expression [0-9]+ is expected to be converted to an integer constant whose value is 0.

In the absence of a matching regular expression to identify constants belonging to a specific type, the types of constants would typically be inferred by the typer following parsing of the original document. The names of any inferred, typer-specific types must not conflict with those reserved in PrimitiveTypes. Note however that the concrete syntax presented in this thesis does not provide any facility for declaring types. The reason for this is that the typer is largely considered beyond the scope of the thesis. Type information for the TypedAST is managed by a TypeMapper and TyperBSOT class which suffice for the purposes of the thesis. The compromise is that types for non-trivial constants and variables must be pre-declared prior to parsing the original document. Failing to pre-declare a non-trivial constant or variable type will result in a type-error being reported during generation of the AST by the parser.

6.0.1 Dealing with Naming Conventions

Up to this stage in the translation process, there have been few restrictions on the names of variables, constants and symbols stored in the AST. Other than recognizing and reserving numeric constants and conventional symbols for arithmetic operations, the document may contain an infinite number of names chosen by its writer. It is recognized however that when a specific oracle is targeted, some user-chosen names may conflict with the naming conventions enforced by the target oracle. A utility class called TokenizedVariableNames is available to assist in this situation. It chooses a straightforward naming convention that is expected to be supported by the majority

of oracles; each of the user-defined variable names is mapped to the tokenized name. When the AST is converted, the tokenized name can be obtained at any time using the user-defined name as a key.

In rare exceptions when the naming conventions of TokenizedVariableNames have a conflict with those of the oracle, the TokenizedVariableNames class can serve as a template that can be extended to implement a naming convention specific to the offending oracle.

6.1 Adapter Interface

This section describes details of the functions that are important to the interface between the first-order translator of an AST and the target oracles.

```
final def notImplemented[T] : Try[T] = Failure(new Exception("not implemented"));
final def notAvailable[T] : Try[T] =
                              Failure(new Exception("the required resource is not available"));
final def unsupportedNode[T] : Try[T] = Failure(new Exception("unsupported node"));
final def illegalName[T] : Try[T] = Failure(new Exception("illegal name"));
final def illegalArgument[T] : Try[Seq[T]] =
                                    Failure(new Exception("error converting argument"));
final def illegalArgumentList[T] : Try[T] = Failure(new Exception("illegal arguments"));
final def undefinedType[T] : Try[T] = Failure(new Exception("untyped argument"));
final def generalFailure[T](message : String) : Try[T] = Failure(new Exception(message));
```

Table 6.0: Adaapter Interface Error Reporting

6.1.0 Scala Interface Trait

The design of the prover interface using Scala makes use of Scala's support for parametric polymorphism and the monadic Try type. In the descriptions of functions used below, the symbol T is a parametric placeholder referring to the native AST form. Concrete adapters will define this parameter to match the specific type native to their oracle.

The interface trait defines a number of possible failure exceptions for this purpose, as listed in the Table 6.0.

6.1.1 Conversion

Once an AST has been translated and typecast to any concrete class derived from ASTNode, it can be converted to the language of the oracle by passing it to the convertTree function:

```
convertTree[N <: ASTNode](ast : N) : Try[T]
```

6.1.2 Session Management

The Adapter interface also defines the means by which the server can initiate, terminate and manipulate running sessions of the oracles it spawns:

```
checkSession() : Boolean; //ABSTRACT
```

6.1.3 Execution of the Query

The reason for converting trees to an Oracle's form in the first place is to be able to pose questions to it. There are two distinct parts of a query: firstly, constraints used to axiomatize a function or definition important to the formula; and secondly, a conclusion which needs to be proven or disproven given the constraints. Once the constraints and the conclusion have been asserted, the query is ready to be executed.

In order to facilitate this, the following functions are declared:

```
assertConstraint(booleanFormula : T)
negateConclusion[N <: ASTNode](conclusion : N) : Try[T]
isFormulaSatisfiable(booleanFormula : T) : Option[Boolean];
```

6.2 Implementation Details

This section describes details of the functions that facilitate the mapping of specific AST trees to ASTs native to each third-party oracle. This information is provided as a reference for future developers and is not required to use a fully developed concrete adapter.

6.2.0 Built-in Semantics For Common Constants

Constants and symbols defined as part of Boolean and arithmetic theory remain in the language. These include the constants representing true and false, conjunction and disjunction, identity and inequality, addition and subtraction, and multiplication and division. Special keywords representing these built-in operations are shared between

the ASTNormalForm and the interfaces to the various oracles. Table 6.1 lists the entry points in the adapter interface support conversion of well-formed Boolean and arithmetic expressions to the target AST of the oracle.

atotFALSE() : Try[T] ;
atotTRUE() : Try[T] ;
atotNOT(term : T) : Try[T] ;
atotAND(lhs : T, rhs : T) : Try[T] ;
atotOR(lhs : T, rhs : T) : Try[T] ;
atotIMPLIES(lhs : T, rhs : T) : Try[T] ;
atotINT(i : Int) : Try[T] ;
atotNEGATE(term : T) : Try[T] ;
atotSUM(lhs : T, rhs : T) : Try[T] ;
atotDIFFERENCE(lhs : T, rhs : T) : Try[T] ;
atotPRODUCT(lhs : T, rhs : T) : Try[T] ;
atotQUOTIENT(lhs : T, rhs : T) : Try[T] ;
atotREMAINDER(lhs : T, rhs : T) : Try[T] ;
atotISLESSTHAN(lhs : T, rhs : T) : Try[T] ;
atotISNOGREATERTHAN(lhs : T, rhs : T) : Try[T] ;
atotISIDENTICALTO(lhs : T, rhs : T) : Try[T] ;
atotISNOLESSTHAN(lhs : T, rhs : T) : Try[T] ;
atotISGREATERTHAN(lhs : T, rhs : T) : Try[T] ;
atotIF(cond : T, ifClause : T, elseClause : T) : Try[T] ;
atotISREFINEDBY(lhs : T, rhs : T) : Try[T] ;

Table 6.1: Adapter Interface Default Conversion Methods

6.2.1 Axiomatization of User-defined Functions

The constants defined above have predefined type and semantics. In addition to the above, any fully applied application of user-defined functions is also supported. The type of these user-defined functions as inferred by the type-checker must be consistent. The interpretation of the functions are subject to the user's axiomatization of the

functions specification. In order to support this, the prover interface allows the user to bind a variable in a tree and optionally quantify over it either existentially or universally. The input ASTNode is translated to the oracles format as part of this process:

atotBIND(lambda : ASTNode) : Try[T] ;

atotEXISTS(typedBinder : ASTNode, lambda : ASTNode) : Try[T] ;

atotFORALL(typedBinder : ASTNode, lambda : ASTNode) : Try[T] ;

The arguments of the expressions inevitably contain constants, variables and, depending on the state space model adopted for the problem, state variables:

atotConstant(name : ASTName, ty : Type) : Try[T]

atotVariable(name : ASTName,ty : Type) : Try[T] ;

atotStateVariable(name : ASTName,ty : Type) : Try[T] ;

Related to the conversion of constants are issues surrounding the reserved names and naming rules held by specific oracles. Where necessary, the prover interface supports conversion of arbitrarily chosen user-defined manes for variables to a systematically chosen name that is guaranteed not to conflict with the reserved names of the oracle.

6.2.2 Uncurrying of Functions

Whether working with built-in functions or user-defined ones, all functional applications processed by the prover interface must be fully applied and must match the type and signature of the function. In order to ensure this is the case, curried functions

are converted to a first-order, fully applied uncurried form:

```
convertCurriedFunctionToArityN( fn : ASTNode,arguments : Seq[ ASTNode ],
  returns : Type) : Try[T];
```

ASTNodes for the operator and the arguments are passed to the conversion function along with a Type representing the function's return type. This allows the fully applied expression to be typed and converted to the application of a function with arity n.

6.3 Summary

This chapter describes how the code implemented as part of the thesis uses the object-oriented features of Scala to support multiple oracles derived from an generic prover interface. The generic prover interface provides core functionality by supporting ASTNodes with a first-order logic type. Future chapters will show examples in which queries sent to the server are evaluated by real oracles extended from the generic prover interface.

Chapter 7

Using SMT Solvers as Oracles

If the signature of the states has a manageable number of state variables, it is possible to implement a state function that maps the state variables of each state to a distinct and unique set of global variables. This allows the query to avoid quantification over states and keeps it in the form of a first-order formula. The problem of solving the query is thereby reduced to a question about whether a valuation exists for the global variables that acts as an example or counter-example for the assertions. Questions of this nature are solved by automated SMT solvers which find true valuations of expressions that include quantifiers, integers, real-numbers and matrixes and their types. In this chapter the use of an SMT solver to answer queries expressed in the language of SIMPPLE is described and results are presented.

7.0 SMT Solvers in General

Once a derivation has been written in first-order, it needs to be determined if the derivation is valid. There is a direct relationship between validity and satisfiability.

"Validity is about finding a proof of a statement; satisfiability is about finding a solution to a set of constraints... Thus, to check [that a formula] is valid (i.e., to prove it), we show its negation to be unsatisfiable."[Research, 2014]

The question of whether a formula is satisfiable is a classic decision problem known as SAT: "given a propositional formula in n variables, is there a valuation for each of the values such that the formula evaluates to true?" Although SAT is NP-complete, automated SAT solvers exist which can solve some practical instances of SAT. SMT solvers augment SAT solvers by including first-order theories about quantifiers, integers, real-numbers and matrixes and their types. A generic syntax for an SMT solver is defined by the SMT-LIB initiative which promotes the adoption of common languages and interfaces for SMT solvers. Examples of SMT queries presented in this thesis adhere to the 2010 version 2.0 standard of the SMT-LIB specification.

Most SMT solvers are based on an implementation of the Davis–Putnam–Logemann–Loveland (DPLL) algorithm. An SMT solver will respond to a query in one of three ways. Either it will compute a valuation that proves the expression can evaluate true, it will prove that no solution exists, or it will be unable to determine the satisfiability of the expression. When the result is wrapped in a Scala Option return value, the three values are mapped to Some(true), Some(false), or None.

When an SMT solver concludes a formula is satisfiable, it yields a single valuation out of a possibly infinite set. To use an SMT solver to prove a statement that the user believes to be universally valid, the formula needs to be negated. SMT solvers need only produce a single counter-example that proves the assertion false. If a valuation satisfies the negated formula, then at least one exception exists which disproves the universal validity of the statement. On the contrary, the formula is universally valid

if its negation is found to be unsatisfiable. Depending on whether the SMT solver is able to find a counter-example, there are four possible scenarios as shown in Table 7.0.

Formula is Universally Valid	Negated Formula is Satisfiable	Counter-Example Exists	Counter-Example Found	Scala Result
False	True	True	True	Some(true)
False	True	True	False	None
True	False	False	False	None
True	False	False	False	Some(false)

Table 7.0: SMT Outcome Scenarios

The logic of SMT is that of many-sorted first-order logic with equality. In many-sorted first-order logic with equality, variables, functions and constants differ only in terms of their arity and distinctness. Constants and variables are semantically equivalent to functions of arity 0; constants differ from variables in that constants are either distinct from one another or redundant, while variables must necessarily equate to exactly one constant. Functions have arity greater than 0 and must be fully applied to have value. The properties of distinctness and arity are expressible in SMTLIB 2.0 so there is no strict requirement to distinguish amongst constants, variables and functions in an SMT solver.

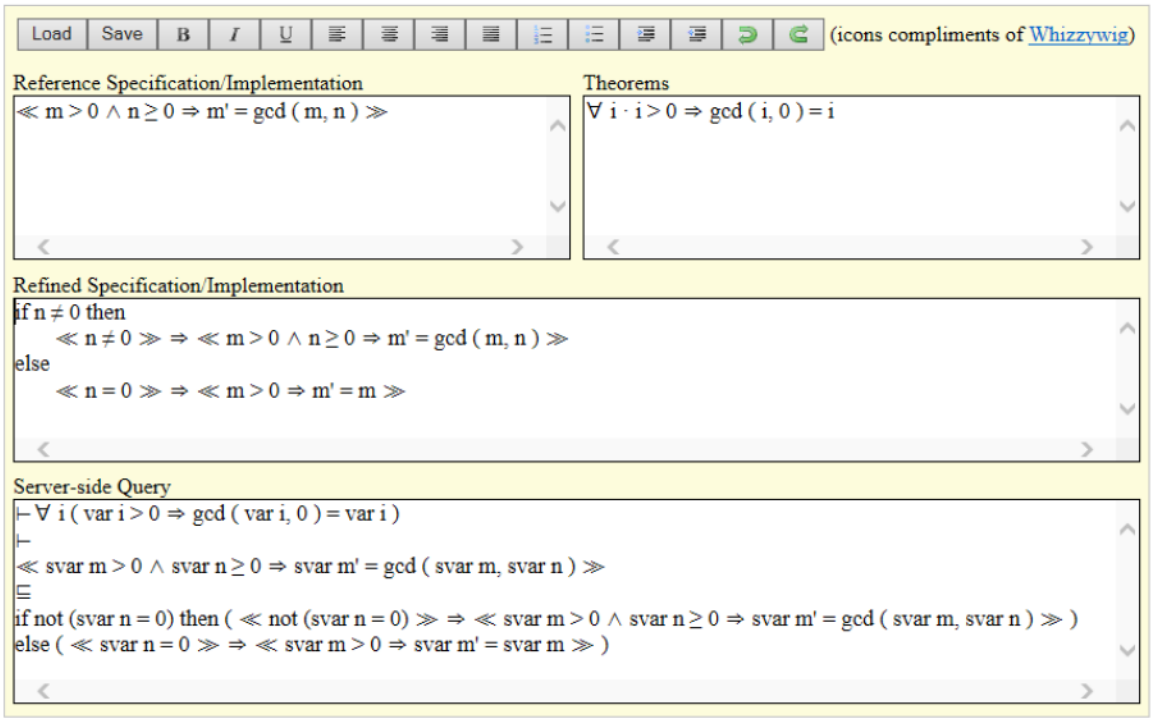


Figure 7.0: Trivial Case in Editor Mockup

7.1 Verification of a Refinement Step

In the example shown in Figure 7.0 the user has applied rules that allow them to solve the trivial case where the second parameter to the GCD function is 0. In order to verify that the original specification is not limited by the proposed implementation, a query asking whether the previous step in the document's history is refined by the current step needs to be sent to the server. A read-only window allows the display of the server-side query for diagnostics and development purposes. Such queries can also be edited and sent directly to the server.

Upon receiving and parsing the query, the server has a sequence of ASTs for the document being edited, including theorems that the user feels relevant to their derivations. Enroute to the prover interface, the AST is converted to a first order

equivalent and program theory is used to convert ASTNodes that have a predefined meaning. These nodes include higher-order specifications and programming instructions such as alternation, assignment and state-variable declaration. To complete the processing of the query, the server must pass it to an external oracle by extending the server's generic prover interface to produce a customized adapter.

7.1.0 Conversion of the AST

The translated AST is sent to the convertTree call of the generic prover interface, which processes the AST in one of two ways:

0. if the oracle has an API, the AST is passed it to the oracle using the API.

1. alternatively, the translator can convert the translated AST to the concrete syntax for the oracle.

7.1.0.0 Interfacing to SMT using an SMT Pretty Printer

A critical goal of the thesis is to integrate the server-side with an efficient SMT solver. One option for doing so is to pretty print[0] the SMT query to a file and execute and analyze the return codes output by a console-based solver. Listing 7.0 illustrates the SMT program that corresponds to the query of Figure 7.0.

Because an SMT solver is being used to show the validity of a formula, the negation of the formula is checked and the SMT solver is expected to determine the formula unsatisfiable.

[0]Pretty printing is the inverse process of parsing.

```
(declare-fun gcd (Int Int) Int)
(assert (forall ((i Int)) (=> (> i 0) (= (gcd i 0) i))))
(assert
    (not
        (forall ((_allOthers Int))
        (forall ((_allOthers_p Int))
        (forall ((m Int))
        (forall ((m_p Int))
        (forall ((n Int))
        (forall ((n_p Int))
            ( =>
                (if (not (= n 0))
                    (=> (not (= n 0)) (=> (and (> m 0) (>= n 0)) (= m_p (↩
↪gcd m n))))
                    (=> (= n 0) (=> (> m 0) (= m_p (gcd m 0))))
                )
                (=> (and (> m 0) (>= n 0)) (= m_p (gcd m n)))
                )
            )
        )))))
    )
)
(check-sat)
```

Listing 7.0: Trivial Case of GCD

7.1.0.1 ScalaZ3 JNI Interface to Z3 Java API

Rather than pursuing the pretty-printed option, the preferred route for interfacing to an SMT solver is by using a Scala API. Z3 [de Moura and Bjørner, 2008], an SMT solver developed by Microsoft as part of its Research in Software Engineering initiative, was chosen as the target oracle because:

0. it has been applied effectively to similar problems in the past, in particular to verification condition checking; and

1. a JNI interface called ScalaZ3 is readily available.

2. ScalaZ3 makes Z3 accessible from the Scala language. ScalaZ3 was developed through the Laboratory of Automated Reasoning and Analysis (LARA) at the Swiss *Ecole Polytechnique Federale de Lausanne* (EPFL).

The query that matches Listing 7.0 was generated by the server-side algorithms through calls to the generic prover interface described in Chapter 6. Calls specific to ScalaZ3 were customized as part of the development of the ScalaZ3 adapter described in Section 7.3. Using ScalaZ3 as the target oracle, the query of Figure 7.0 was sent to Z3 and verified.

7.2 Case Studies

In this section, we look at three case studies that successfully use an SMT solver to verify a series of refinements from initial specification of a document to final implementation.

7.2.0 General Swap

Earlier in the thesis, the scenario was discussed in which the client-side editor contains the program:

```
declare t := svar a in (
    svar a := svar b ;
    svar b := svar t
)
```

This string is considered first for a number of reasons:

```
DECLARE
        t :=
              svar a
in (
      assign  svar a := svar b ;
      assign  svar b := svar t

)
```

```
StateVarDecl(
                Plain("t"),
                StateVar("a"),
                Apply(          Apply(  Const(";"),
                                        Assign(StateVar("a"),StateVar("b"))
                                ),      Assign(StateVar("b"),StateVar("t"))
                )
)
```

Figure 7.1: General Swap AST

- it is concise;
- it has no arithmetic operations;
- it is immediately identifiable as a program that swaps a with b; in other words, it refines the specification $\langle var\ a' = var\ b \wedge var\ b' = var\ a \rangle$

7.2.0.0 General Swap Implementation

To construct the query, the input string for the implementation needs to be parsed into the AST format of Chapter 2, with translations to first order logic as described in Chapter 5, and then passed to the prover interface as described in Chapter 6.

With the possible exception of the semicolon, the corresponding AST representation of the input string should be intuitively clear. For added clarity, the original input string and the output AST is placed side-by-side in 7.1 to highlight the correspondence between the concrete and abstract form.

7.2.0.1 General Swap Specification

In the grammar of the concrete syntax, the specification of the general swap problem is written as $\langle\langle var\ a' = var\ b \wedge var\ b' = var\ a \rangle\rangle$. Figure 7.2 shows the original input

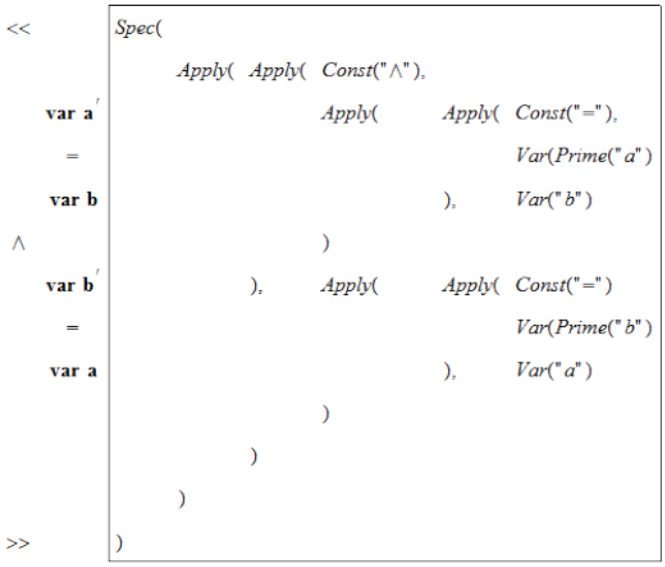


Figure 7.2: Parsed General Swap Specification

string is placed in the left and the corresponding AST on the right to highlight the correspondence between the concrete and abstract form.

Currying in this example can be seen to produce unnatural looking transcripts. For this reason, the remaining examples will discuss only the relationship between the concrete syntax and the output of the SMT solver. The reader should understand that the concrete syntax is for presentation and that the document is stored internally by parsing the concrete syntax into the AST presented in Chapter 2.

7.2.0.2 Verification of General Swap Implementation

The belief that an implementation meets the requirements of its user is for all intents and purposes a theorem. Developers expect their users to accept the existence of an implementable program that appears to behave according to the specification as proof that the work is complete. The concrete syntax presented in this section is a

language that formalizes this contract. This proof is expressed in the concrete syntax as a derivation:

$\vdash$

$\langle\langle svar\ a' = svar\ b \wedge svar\ b' = svar\ a\rangle\rangle$

$\sqsubseteq$

declare svar t := svar a in (

    svar a := svar b;

    svar b := svar t

)

The above query represents a coarse-grained formal proof, moving directly from specification to implementation. A fine-grained formal derivation of the program involves application of substitution laws of assignment one law or rule at a time. When using SMT solvers as oracles, the server accepts either style of proof.

Details of the conversion of the query to SMT are logged along with the final result of the SMT solver. The query above is converted to an equivalent AST representable by the concrete syntax of SMTLIB. This portion of the log is shown in Listing 7.1. In response to the query the Z3 solver returns the result of Some(false). As shown in Table 7.0, this indicates that the negated form of the refinement is unsatisfiable, thereby confirming validity of the assertion that the implementation refines the specification.

Introducing a bug, such as the one in Listing 7.2 causes the query to return Some(true). Altering the specification in various ways, such as that depicted in Listing 7.3, is likewise detected by the solver as failing the refinement. As indicated

```
(not (forall (_allOthers Int)
(forall (_allOthers_p Int)
(forall (x Int)
(forall (x_p Int)
(forall (y Int)
 (forall (y_p Int)
   (=> (exists (t_p Int)
         (exists (_allOthers_d Int)
           (exists (x_d Int)
             (exists (y_d Int)
               (exists (t_d Int)
                 (and (and (= _allOthers_d _allOthers)
                           (and (= x_d x)
                                (and (= y_d y) (= t_d y))))
                      (exists (_allOthers_d_d Int)
                        (exists (x_d_d Int)
                          (exists (y_d_d Int)
                            (exists (t_d_d Int)
                              (and (and (= _allOthers_d_d
                                           _allOthers_d)
                                        (and (= x_d_d x_d)
                                             (and (= t_d_d t_d)
                                                  (= y_d_d x_d))))
                                   (and (= _allOthers_p
                                           _allOthers_d_d)
                                        (and (= y_p y_d_d)
                                             (and (= t_p t_d_d)
                                                  (= x_p t_d_d))))))))))))))
       (and (= x_p y) (= y_p x)))))))))))
```

Listing 7.1: Equivalent Server-side Query in SMTLIB

in Table 7.0, the SMT solver can provide a counter-example in these cases.

7.2.1 Algebraic Swap

Earlier in the thesis, the scenario was discussed in which a user opens a document containing the program given by Listing 7.4. After editing, the user ends up with the

```
⊢
<< svar a' = svar b ∧ var b' = svar a >>
⊑
declare svar t := svar a in (
    svar a := svar a;
    svar t := svar b
)
```

Listing 7.2: Example of a Failed Implementation

```
⊢
<< svar a' = svar b ∧ var b' = svar a ∧ var c' = svar a >>
⊑
declare svar t := svar a in (
    svar a := svar b;
    svar b := svar t
)
```

Listing 7.3: Example of an Incomplete Implementation

```
a := b + a;
b := a − b;
a := a − b
```

Listing 7.4: Initial Document

program seen in Listing 7.5. When using SMT solvers as oracles, the server is able to compare programs to program (see Listing 7.6) in the same manner that it compares programs to specifications. The SMT solver applies its theorems of integer theory during the processing of this query and successfully verifies the refinement relation between the two operations.

It is important to note that the theory of integers used bySIMPPLE (and the SMT solver) assumes unbounded integers. This establishes that the programs are

```
declare t := a in (
a := b
b := t;
)
```

Listing 7.5: Edited Document

```
⊢
svar a := svar b + svar a;
svar b := svar a − svar b;
svar a := svar a − svar b
⊑
declare svar t := svar a in (
    svar a := svar b;
    svar b := svar t
)
```

Listing 7.6: Server-side Query

```
⊢
declare svar t := svar a in (
    svar a := svar b;
    svar b := svar t
)
⊑
svar a := svar b + svar a;
svar b := svar a − svar b;
svar a := svar a − svar b
```

Listing 7.7: Equivalence for Unbounded Integers

equivalent when the state variables are declared to be unbounded Integer types. It also means the reverse relationship verifies as seen in Listing 7.7.

7.2.2 Formal Derivation of GCD

In Chapter 2 of the thesis, the formulation of a problem statement for computing the greatest common divisor and its trivial solution was introduced as a demonstration of the method programming by stepwise refinement. The GCD algorithm can be derived using fine-grained application of rules that allow each step to be compared by a human reader for correctness. In this section, an adaptation of the fine-grained formal proof of the GCD [Norvell, 2012] is presented in such a fashion with a brief discussion of the rules applied by each step. The code samples are excerpts from the log generated by the Scala software written in the course of the thesis.

7.2.2.0 Alternation Law

The case creation law[Hehner, 2014] states that for any specification P, and any Boolean expression b, P can be rewritten as $if\ b\ then\ \ (b \Rightarrow P)\ \ else\ \ (\neg b \Rightarrow P)$. SIMPPLE uses the closely related Alternation law, in which angle brackets convert the Boolean expression to the higher-order type of a specification; hence one writes:

$if\ b\ then\ (\langle\ b\ \rangle \Rightarrow P)\ \ else\ \ (\langle\ \neg b\ \rangle \Rightarrow P.)$

In this form, one can approach the derivation in a case-wise fashion. Any step that refines the clause $\langle\ b\ \rangle \Rightarrow P$ can be used in the case when b evaluates true; likewise, any step that refines the clause $\langle\ \neg b\ \rangle \Rightarrow P$ can be used in the case when b evaluates false.

The first step in the derivation of the GCD algorithm is to apply this to the specification for the GCD as seen in Listing 7.8. The discussion of the proof continues in the following sections, first for the trivial case where $n = 0$ and then for the more

```
\Theorem:
<< svar m > 0 and svar n >= 0 => ( svar m' = gcd (svar m, svar n)) >>
\isRefinedBy
if not (svar n = 0)
    then
        ( << not (svar n = 0) >> =>
            << svar m > 0 and svar n >= 0 => ( svar m' = gcd (svar m, svar n)↩
↪) >> )
    else
        ( << not (not (svar n = 0)) >> =>
            << svar m > 0 and svar n >= 0 => ( svar m' = gcd (svar m, svar n))↩
↪ >> )
```

Listing 7.8: Application of Alternation Law

```
\Theorem:
<< (svar n = 0) >> => << svar m > 0 and svar n >= 0 => ( svar m' = gcd (↩
↪svar m, svar n)) >>
\isRefinedBy
<< (svar n = 0) =>
        (svar m > 0 and svar n >= 0) => ( svar m' = gcd (svar m, svar n)) >>
```

Listing 7.9: Elimination of Guarded Expression

```
\Theorem:
<< (svar n = 0) =>
        (svar m > 0 and svar n >= 0) => ( svar m' = gcd (svar m, svar n)) >>
\isRefinedBy
<< ((svar n = 0) and (svar m > 0 and svar n >= 0)) => ( svar m' = gcd (svar m,↩
↪ svar n)) >>
```

Listing 7.10: Shunting

general case

```
0  \Theorem:
   << ((svar n = 0) and (svar m > 0 and svar n >= 0)) => ( svar m' = gcd (svar m,↩
   ↪ svar n)) >>
   \isRefinedBy
   << ((svar n = 0) and (svar m > 0 )) => svar m' = gcd (svar m, svar n) >>
```

Listing 7.11: Simplification

7.2.2.1 Elimination of Guarded Expressions

When two specifications are combined using the higher-order implication operator => they produce a new specification inside which the LHS operand implies the RHS using ordinary Boolean implication. This definition is applied to the clause where $n = 0$ in Listing 7.9

7.2.2.2 Shunting

Shunting asserts the logical equivalence between the expressions $P \Rightarrow Q \Rightarrow R$ and $P \wedge Q \Rightarrow R$. This rule in applied in the next revision in Listing 7.10

7.2.2.3 Simplification

The expression $n \geq 0$ includes the expression $n = 0$ and is the weakest of the two expressions. The antecedent of the implication can therefore be simplified to Listing 7.11 using the equivalence $(n = 0 \wedge n \geq 0) = (n = 0)$.

The net effect of the refinements made to the specification up until this point is to simplify the precondition of the case when $n = 0$ as much as possible. This is a common strategy in stepwise refinement and it will be seen again when the more general case of $n \neq 0$ is refined.

```
0  \Theorem:
   << ((svar n = 0) and (svar m > 0 and svar n >= 0)) => ( svar m' = gcd (svar m,↩
   ↪ svar n)) >>
   \isRefinedBy
   << ((svar n = 0) and (svar m > 0 )) => svar m' = svar m >>
```

Listing 7.12: Application of Trivial Solution

7.2.2.4 Evaluation of Trivial Solution

The case when $n = 0$ has now been reduced to a specification that defines the pre-conditions and the desired output. At each stage, the query has been sent to the server and has been converted to Z3. The result of the query in all cases has been Some(false). As can be seen from Table 7.0, this indicates that the negated form of the refinement is unsatisfiable, thereby confirming validity of the assertion that each step in the derivation refines the previous step.

In the case of the GCD, a trivial solution occurs when one of the inputs is 0 and the other is a positive number: these conditions match those now in the simplified form of the specification. Since zero can be divided by any number, the largest number which divides the other number is therefore the solution. Since the other number is positive number it divides itself and is the largest number to do so; that makes it the solution. We can therefore solve the trivial case by setting $m' = m$ as seen in Listing 7.12.

Sending the above query to the server results in the first failure from the Z3 SMT solver. Instead of reporting Some(false), it reports Some(true). This indicates that an interpretation exists for the uninterpreted function GCD that makes it possible for the proposed solution ($m' = m$) not to be equivalent to the desired solution

```
0  \Theorem:
   forall i (
       var i > 0 =>
           gcd (var i, 0) = var i
   )
```

Listing 7.13: Theorem for GCD(i,0)

```
0  \Theorem:
   << ((svar n = 0) and (svar m > 0 )) => svar m' = svar m >>
   \isRefinedBy
       skip
```

Listing 7.14: Introduction of *skip*

($m' = \gcd(m, 0)$). Since in the intended interpretation of the GCD function, it is a law that $\forall i \cdot i > 0 \Rightarrow \gcd(i, 0) = i$ a means of restricting Z3 from using any interpretation that does not satisfy this law is required. To do so, the theorem of Listing 7.13 must be added to the document.

With the addition of the above theorem, the document is tested again. Theorems such as the one above are extracted from the document and sent to the Z3 SMT solver in order to augment its background theories and to constrain the interpretations it can use for uninterpreted functions. The remaining queries are then tested; now the SMT solver is able to verify that, given what it knows about the GCD algorithm, the proposed refinement is valid.

7.2.2.5 Introduction of Skip

The final step for generating a program from a specification is to refine the specification by statements that consist only of computer instructions. The simplest of

these instructions is the "no-operation" or *skip* instruction. As a result of our formal derivation, the system is able to verify that in the case when $n = 0$ the *skip* instruction is a suitable implementation as seen in Listing 7.14.

7.2.2.6 Reduction of General Case to a Simpler Case

Euclid's GCD algorithm provides a proof that in the general case, the solution of the GCD is equal to the GCD of one number and the modulo remainder of the larger number. Since modulo arithmetic requires a positive divisor, this corresponds to the theorem: $\forall i \cdot \forall j \cdot j > 0 \Rightarrow \gcd(i, j) = \gcd(j, i\%j)$. By adding this theorem to the document, a recursive solution of the GCD can be verified using a method similar to that for the trivial case. This time, all steps are combined into a single equational proof, that simplifies the general case to its precondition and its output requirements, and then converts the output assignment to a recursive assignment instruction. The server-side query for this is shown in Listing 7.15.

The Z3 SMT solver is able to verify each step. Running the example provides a log of the equivalent SMTLIB concrete syntax and the Z3 response Some(false) for each refinement, indicating validity of the affirmative form of the query.

7.2.2.7 While Law

The theory of stepwise refinement of programs includes support for recursive calls and loops. For example, the While Law states that, subject to some conditions, a query in the form $g \sqsubseteq \mathit{if}\ b\ \mathit{then}\ (h; g)\ \mathit{else}\ \mathit{skip}$ can be rewritten as $g \sqsubseteq \mathit{while}\ b\ \mathit{do}\ h$ [Norvell, 2012]. The use of looping instructions has not been directly addressed in this thesis, but the reader should recognize that, having established the previous theorems within

```
\Theorem:
forall i forall j (
      var j > 0 =>
         gcd (var i, var j) = gcd (var j, var i % var j )
\Theorem:
  << not (svar n = 0) >> => << svar m > 0 and svar n >= 0 =>
         ( svar m' = gcd (svar m, svar n)) >>
\isRefinedBy
  << (not (svar n = 0) ) =>
         (svar m > 0 and svar n >= 0) => ( svar m' = gcd (svar m, svar n)) >>
\isRefinedBy
  << (not (svar n = 0) and (svar m > 0 and svar n >= 0)) =>
         ( svar m' = gcd (svar m, svar n)) >>
\isRefinedBy
  << (svar m > 0 and svar n > 0) => (svar m' = gcd (svar m, svar n)) >>
\isRefinedBy
  << (svar m > 0 and svar n > 0) => ( svar m' = gcd (svar n, svar m % svar n)) ↩
↪>>
\isRefinedBy
  declare t := svar n
  in (
         assign svar n := svar m
          assign svar m := svar t ;
  );
  << svar m > 0 and svar n >= 0 => ( svar m' = gcd (svar m, svar n)) >>
```

Listing 7.15: Recursive Solution to General Case

the document, the GCD specification has been proven to be refined to this form as seen in Listing 7.16.

The conditions of the While Law are required to ensure that the loop eventually terminates. In the case of the GCD, since $n > (m \ \% \ n) >= 0$, the modulo arithmetic of the general case works to reduce the problem to the trivial case of $n = 0$ in at most n iterations. As proof of termination, we can add an iteration counter state variable *svar c* and require that as part of the specification, the final value of *svar c* be no

```
\Theorem:
  << svar m > 0 and svar n >= 0 => (svar m' = gcd (svar m, svar n) ) >>
\isRefinedBy
  if not (svar n = 0)
  then (
   declare t := svar n
   in (
         assign svar n := svar m % svar n ;
         assign svar m := svar t ;
   );
   << svar m > 0 and svar n >= 0 => ( svar m' = gcd (svar m, svar n)) >>
  )
  else
   skip;
```

Listing 7.16: Final GCD Algorithm

greater than n above its initial value; that is: $\langle\langle c' <= c + n \rangle\rangle$

The modified algorithm given in Listing 7.17 includes proof of termination and passes the test for validity when using the Z3 SMT solver as the oracle. In the presence of this proof of termination, the GCD implementation can be rewritten with while loops as shown in Listing 7.18.

One closing comment: changing the termination condition from $\langle\langle c' <= c + n \rangle\rangle$ to $\langle\langle c' < c + n \rangle\rangle$ did not pass the test for validity. Rather than returning, the Z3 SMT solver enters an infinite loop. It is expected from Table 7.0 that less exhaustive SMT solvers would return None in response to the query, indicating an inability to prove or disprove the existence of a counter-example. To accommodate for such possibilities, time-outs are recommended to be added to the prover interface. Support for additional SMT solvers should also be considered; the query can then be sent to multiple solvers at once and the first solver to return with a definitive response can signal the termination of the others.

```
\Theorem:
  << svar m > 0 and svar n >= 0 => ( (svar c' <= (svar c + svar n) and (svar ↩
↪m' = gcd (svar m, svar n))) >>
\isRefinedBy
  if not (svar n = 0)
  then (
   declare t := svar n
   in (
        assign svar n := svar m % svar n ;
        assign svar m := svar t ;
   );
   assign svar c := svar c + 1;
   << svar m > 0 and svar n >= 0 => ( (svar c' <= (svar c + svar n) and (svar ↩
↪m' = gcd (svar m, svar n))) >>
  )
  else
   skip;
```

Listing 7.17: Proof of Termination

```
\Theorem:
  << svar m > 0 and svar n >= 0 => ( (svar c' <= (svar c + svar n) and (svar ↩
↪m' = gcd (svar m, svar n))) >>
\isRefinedBy
  while (svar n = 0)
  do (
   declare t := svar n
   in (
        assign svar n := svar m % svar n ;
        assign svar m := svar t ;
   );
   assign svar c := svar c + 1;
  )
```

Listing 7.18: Alternate GCD Algorithm

7.3 Implementation Details

This section describes details of the functions that adapt an AST to a form that can be executed by Z3 by way of the ScalaZ3 API. This information is provided as a

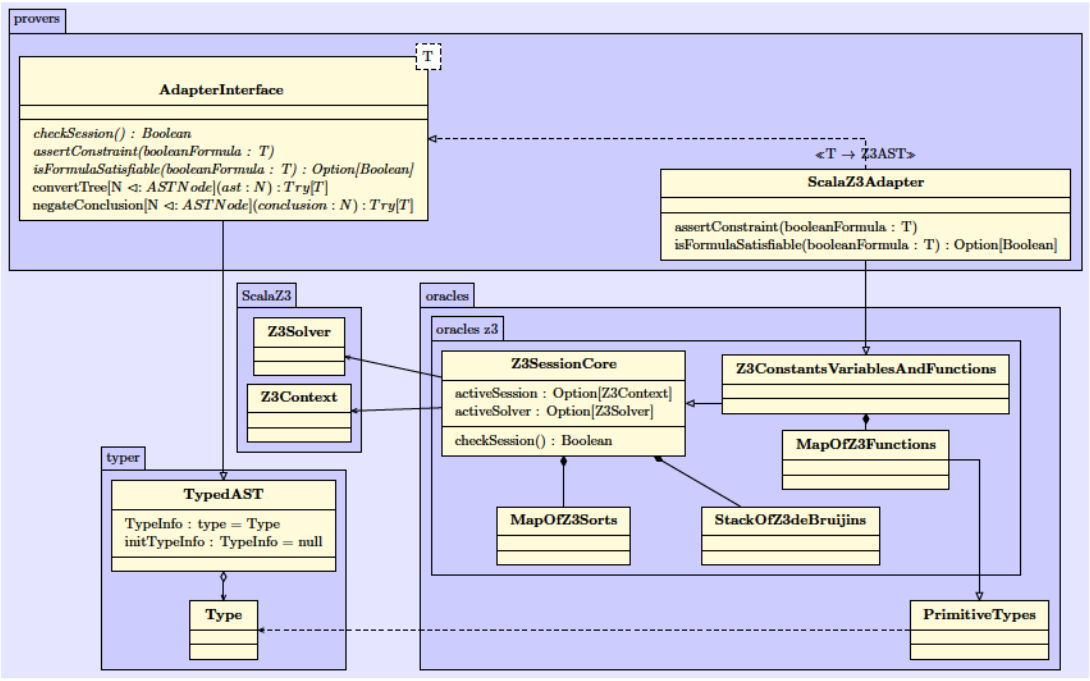


Figure 7.3: ScalaZ3Adapter Class Diagram

reference for future developers and is not required to use a fully developed concrete adapter. Figure 7.3 shows the classes involved.

7.3.0 Session Management

The following functions allow the adapter to check whether a session to z3 is open, to open and reset resources used by the session, and to close and free resources allocated to the session:

- def checkSession() : Boolean
- def open() : Boolean
- def close()

7.3.1 Type Declaration

Types in SMTLIB are referred to as Sorts. By default, z3 recognizes boolean and integer sorts. Any other type must be declared inside z3 before its use. The following functions allow the adapter to retrieve the sort from z3 if it has been declared, and optionally to declare it if it does not already exist.

- def findSortInContext(sortRequired : String) : Option[Z3Sort]
- def ensureSortInContext(sortRequired : String) : Z3Sort

7.3.2 Constants, Variables and Functions

The implementation of the interface to z3 processes constants, variables and functions under the umbrella of one class, and handles the ASTNodes: Apply, Const, and Var. It distinguishes between constants and functions, the latter of which must always be applied to at least one argument. In order to ensure there are no naming, arity or typing inconsistencies, it enumerates constants, variables and functions as it encounters them in the AST, calling a routine that performs the necessary checks. These routines include:

- def findTypedConstantInContext(name : String, sortRequired : String) : Option[Z3AST]
- def ensureTypedConstantInContext(name : String, sortRequired : String) : Option[Z3AST]
- def findTypedVariableInContext(name : String, sortRequired : String) : Option[Z3AST]

- def ensureTypedVariableInContext(name : String, sortRequired : String) : Option[Z3AST]
- def findFunctionInContext(name : String, sortRequired : String, argsRequired : Seq[String]) : (String,Option[Z3FuncDecl])
- def ensureFunctionInContext(name : String, sortRequired : String, argsRequired : Seq[String]) : (String,Option[Z3FuncDecl])
- def findTypedFunctionInContext(name : String, sortRequired : String, argsRequired : Seq[String]) : Option[Z3FuncDecl]
- def ensureTypedFunctionInContext(name : String, sortRequired : String, argsRequired : Seq[String]) : Option[Z3FuncDecl]

7.3.3 Existential and Universal Quantification

Another important element of SMT is the assertion of universally or existentially quantified expressions. Using the ScalaZ3 API, the adapter first builds these as lambda-abstracted formulas; then fresh symbols are created for each of the bound variables and used to assert the appropriate quantification. The following routines are provided in order to meet these requirements:

- def findSymbolInContext(symbolRequired : String) : Option[Z3Symbol]
- def ensureSymbolInContext(symbolRequired : String) : Z3Symbol
- def buildBoundArgumentList (functionSignature : (Seq[String],String)) : (Seq[Z3AST], Seq[(Z3Symbol,Z3Sort)])

7.4 Summary

This chapter describes how the code implemented as part of the thesis uses the object-oriented features of Scala to evaluate server-side queries by targeting the ScalaZ3 SMT solver. UML diagrams describing the implementation are provided along with case studies. A ScalaTest test suite containing the case studies presented here is maintained at svn://tera.engr.mun.ca/simple.proj/trunk/SIMPPLE.

Chapter 8

Using Higher-Order Theorem Provers

The cases studies in this thesis have focussed on oracles that provide feedback about whether one tree stored in AST format is a refinement of another. For these purposes, SMT and first-order logic suffice. That said, the software has been designed with the view to testing significantly more complex queries and formulating complete theories and specifications. Figure 8.0 shows a conceptual prototype for a concrete adapter based around a serializer that pretty-prints[0] a TPS document.

In this chapter we show how the above interfaces support adapters to a popular third-party TPS in order to process the query.

[0]Pretty-printing is the inverse process of parsing. Starting from an AST, the pretty-printer selects a concrete syntax of its choosing.

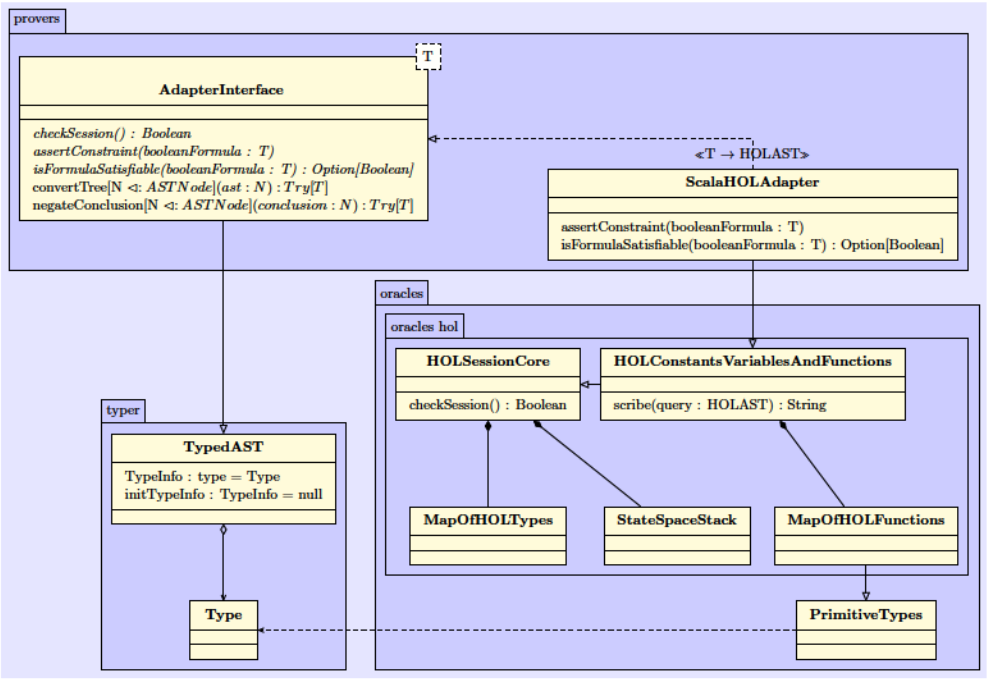


Figure 8.0: ScalaHOLAdapter Class Diagram

8.0 Writing Theories of Computation

It is well-known that compilable source code can be generated directly from logical formulas that describe the intended behaviour of software. Theories in support of this goal, including theories of predicative programming and programming by specification, were developed and well-understood by the mid-1990's. These theories come in a number of forms. Already in this thesis we have seen two closely related forms: one first-order form that views specifications as Boolean formulas and refinement as ordinary implication; and a closely related high-order form that views specifications as a function of two states, and refinement as a preorder between specifications. This distinction is abstracted away when discussing programming instructions, but becomes important when formulating specifications. As an example the theories included in

a NECEC 2010 conference paper[Motty and Norvell, 2010] are reproduced in Table 8.0.

The language and environment for which this thesis is written is that of a browser-based extension for editing of instructions, Boolean formulas, and high-order specifications. In this chapter, a distinction is made between what is supported by the AST, what can be expressed in the language, what theories of computation are employed, and what can be proven.

Term	First-order Interpretation	Higher-order Interpretation
$\|\|\mathbf{skip}\|\|$	$x' = x \wedge y' = y$	$\lambda\, s \cdot \lambda\, s' \cdot (s = s')$
$\|\|E\|\|$	E	$\lambda\, s \cdot E^{x,y}_{sx,sy}$
$\|\|x{:=}E\|\|$	$x' = E \wedge y' = y$	$\lambda\, s \cdot \lambda\, s' \cdot \forall v \cdot\ s'\, v = \|\|E\|\|\, s$ $\lhd v = x \rhd s'\, v = s\, v$
$\|\|\langle S \rangle\|\|$	S	$\lambda\, s \cdot \lambda\, s' \cdot S^{x,y,x',y'}_{sx,sy,s'x,s'y}$
if E **then** F **else** G	$\|\|F\|\| \lhd E \rhd \|\|G\|\|$	$\lambda\, s \cdot \lambda\, s' \cdot \|\|F\|\|\, s\, s' \lhd \|\|E\|\|\ s \rhd \|\|G\|\|\ s\, s'$
$\|\|F; G\|\|$	$\exists \dot{x}, \dot{y} \cdot \|\|F\|\|^{x',y'}_{\dot{x},\dot{y}} \wedge \|\|G\|\|^{x,y}_{\dot{x},\dot{y}}$	$\lambda\, s \cdot \lambda\, s' \cdot \exists \dot{s} \cdot \|\|F\|\|\ s\, \dot{s} \wedge \|\|G\|\|\ \dot{s}\, s'$
$\|\|F \implies G\|\|$	$\|\|F\|\| \Rightarrow \|\|G\|\|$	$\lambda\, s \cdot \lambda\, s' \cdot \|\|F\|\|\ s\, s' \Rightarrow \|\|G\|\|\ s\, s'$
$\|\|F \sqsubseteq G\|\|$	$\forall x, y, x', y' \cdot \|\|G\|\| \Rightarrow \|\|F\|\|$	$\forall s \cdot \forall s' \cdot \|\|G\|\|\ s\, s' \Rightarrow \|\|F\|\|\ s\, s'$

Table 8.0: Interpretations of Commands and Refinement[1]

8.0.0 Supported High-order Formulas

The first of these, what is supported by the AST, is the least restrictive. For example, the AST can store any of the terms from Table 8.0.

[1] In Table 8.0, it is assumed that all commands operate on states whose state variables include only x, and y. Formulas need to be modified accordingly for other states.

8.0.1 Expressible High-order Formulas

As for what can be expressed, the concrete syntax in this thesis intentionally omits lambda expressions. The ability to chain and nest such expressions inside other expressions makes their translation to lower-orders difficult and could not be completed within the scope of this thesis.

Without support for pure (unqualified) lambda expressions in the concrete syntax, many laws required for the high-order theory of computation are excluded. For instance, using lambda notation, the high-order laws of forward and backward substitution are written as shown below:

$$\vdash \forall s \cdot \forall x \cdot \forall z \cdot sub\ s\ x\ z = \lambda y \cdot z \lhd x = y \rhd s\ y$$

$$\vdash \forall f \cdot \forall x \cdot \forall e \cdot forwardSub\ f\ x\ e = \lambda s \cdot \lambda s' \cdot f\ (sub\ s\ x\ (e\ s))\ s'$$

$$\vdash \forall f \cdot \forall x \cdot \forall e \cdot forwardSub\ f\ x\ e = (x := e; f)$$

$$\vdash \forall f \cdot \forall x \cdot \forall e \cdot backwardSub\ f\ x\ e = \lambda s \cdot \lambda s' \cdot f\ s\ (sub\ s'\ x\ (e\ s'))$$

$$\vdash f \sqsubseteq backwardSub\ f\ x\ e; x := e$$

8.0.2 Employable Theories of Computation

The use of the high-order theory of computation has been deferred in favour of the first-order state-space model. This allows use and testing with readily available first-order tools.

The usability of high-order theories of computation is significantly enhanced by enabling lambda abstractions in the concrete syntax, but the employment of these the-

ories is not however dependent on the presence of lambda expressions in the concrete syntax. High-order theories of computation can be added by modifying the Scala code described in chapters 2 to 5 of this theses. This section provides a summary of the required changes.

8.0.2.0 Tree-based substitution with State

The functions which apply the model of state space need to be overridden in order to take full advantage of higher-order types available in theorem proving systems. The following is recommended[Norvell, 2009]

0. StateVar

The state variable is converted to a functional application of the corresponding state and a string representing the name of the state variable. Handling of decorated names needs to be transferred from states to variables: the notation $x_d = Dot(x_{d-1})$ for $d > 0$ and $x_0 = x = Plain(x)$ is used to illustrate this. A function *dotState* is needed to handle intermediate states by converting $.Apply(\ Var(s_d)),\ v\ )$ to $Apply(\ Var(Dot(s_{d-1})),\ v\ )$ where

$$\|StateVar(\ Plain(v)\ )\| = Apply(\ Var(Plain(s)),\ v\ )$$

$$\|StateVar(\ v_d\ )\| = dotState(\|StateVar(\ v_{d-1}\ )\|)$$

$$\|StateVar(\ Prime(v)\ )\| = Apply(\ Var(Prime(s)),\ v\ )$$

1. Application of semicolon to two commands

   (a) The inner commands are translated. To eliminate the possibility of conflict between bound and free variables, an extra dot is appended to existing

dotted states in both terms; this includes any dotted variables bound by a lambda abstraction.

(b) A tree representing the left-most command is applied to the plain state and an existentially bound intermediate (dotted) state; the right-most command is applied to the dotted state and the Primed state. This thereby associates the right-hand side's initial state with the intermediate state constrained by the left-hand side.

$$\| \mathit{Apply}\,(\ \mathit{Apply}\,(\ \mathit{Const}\,(\ ";"\ )\ ,\ \mathit{lhs}\ )\ ,\ \mathit{rhs}\ )\ \| =$$

$$\begin{aligned}
&\mathit{Lambda}\Big(\mathit{Plain}(s), \mathit{Lambda}\Big(\mathit{Prime}(s),\\
&\quad \mathit{Apply}\Big(\mathit{Const}\,(\exists)\,, \mathit{Lambda}\Big(\mathit{Dot}\,(s)\,,\\
&\qquad\qquad \mathit{Apply}\,(\mathit{Apply}\,(\ \ \mathit{Const}\,(\wedge)\,,\\
&\qquad\qquad\qquad\qquad \mathit{scLhs}\,(\|\mathit{lhs}\|))\,,\\
&\qquad\qquad\qquad\qquad \mathit{scRhs}\,(\|\mathit{rhs}\|)) \qquad \Big)\Big)\Big)\Big)
\end{aligned}$$

2. Assignment:

(a) The target of the assignment is translated. Currently the software only supports assignments whose target is a single state variable (e.g. "v"), in which case the state variable is promoted to a Primed variable of the same name, and then translated yielding $\|\mathit{StateVar}(\mathit{Prime}(\text{"}\mathbf{v}\text{"}))\|$.

(b) The expression to be assigned to the target is translated.

(c) There must be no side-effects; this is accomplished by universally quantifying over all variable names. If the variable name matches that of the

target, then the value in the Primed state is equal to the translated expression; otherwise the value in the Primed state must be identical to the value in the Plain state.

$$\| \mathit{Assign}(\ \mathit{StateVar}\ (\ \mathit{Plain}(\ \text{"}\mathbf{v}\text{"}\ ))\ ,\ \mathit{rhs}\ )\ \| =$$

$$\begin{aligned}
&\mathit{Lambda}\Big(\mathit{Plain}(s), \mathit{Lambda}\Big(\mathit{Prime}(s),\\
&\quad \mathit{Apply}\Big(\mathit{Const}\ (\forall)\ , \mathit{Lambda}\Big(w,\\
&\qquad\qquad \mathit{Apply}(\mathit{Apply}(\mathit{Apply}\ (\mathit{Const}\ (\ \text{"}\mathit{ite}\text{"}\ )\ ,\\
&\qquad\qquad\quad \mathit{Apply}(\mathit{Apply}(\mathit{Const}\ (\text{"}=\text{"})\ ,\\
&\qquad\qquad\qquad \mathit{Var}\ (\mathit{Prime}(w))),\\
&\qquad\qquad\qquad \mathit{Var}\ (\mathit{Prime}(\text{"}\mathbf{v}\text{"}))),\\
&\qquad\qquad\quad \mathit{Apply}\ (\|\mathit{rhs}\|\ , \mathit{Plain}\ (s))),\\
&\qquad\qquad\quad \mathit{Apply}\ (\ \mathit{Var}\ (\mathit{Prime}\ (s))\ , w))
\end{aligned}
\qquad \Big)\Big)\Big)\Big)$$

3. StateVar Declaration

   (a) The declaration is unapplied to expose the state variable being declared, its initial value, and the body within which the variable is in scope.

   (b) The type of the expression is inferred and a fresh state is instantiated with type constraints defined by universally quantifying over all variable names. If the variable name matches that of the variable being declared, then the value in the Plain state is equal to the translated expression.

   (c) A tree representing the body is translated and it is applied to the freshly created state and an existentially bound final (primed) state. Within the

scope of the existential operation, assertions are made that for all variable names, Apply(Prime(sigma),variable.name) takes the value of Apply(Prime(tau),variable.name) except in the case for the variable name being introduced by the StateVar Decl ; in that case it takes the value of Apply(Plain(sigma),variable.name).

4. Spec

   (a) The declaration is unapplied to expose the body of the spec.

   (b) The body is translated to a lambda abstraction over two states, thereby capturing the Plain and Primed versions of states appearing in the specification and renaming if necessary to simplify the logic of the conversion. The lambda abstraction is then applied to two arguments: one plain and one primed, representing the states for which the specification is to be evaluated:

$$\|Spec\ (\ Body)\| =$$

$$Lambda\Big(Plain(s),\quad Lambda\Big(Prime(s),\\ Apply(Apply(\|Body\|\ , Var(Plain(s)), Var(Prime(s))))\quad \Big)\Big)$$

8.0.2.1 Built-in Laws and Rules

In addition to updating the state space model, inclusion of keywords into the concrete syntax is also suggested in order to unburden the user from the need to declare these. Essential laws include[Norvell, 2012]:

- One-point Law

- Strengthening
- Monotonicity
- Antimonotonicity
- Erasure Laws
- Substitution Laws
- Alternation Law

8.0.2.2 Translation of State-space Types

The translation of Types is simplified by the method of conversions above; after the translations are applied all queries should evaluate to expressions of BoolType(). To support the addition of user-defined types and improve their readability, a method for annotating variables and expressions with explicit types should be introduced into the concrete syntax.

8.0.3 Provable High-order Theories

Quantification over functions requires a type-system with support for functions. Such type-systems are absent from the SMTLIB language, so SMTLIB is only suitable for fragments of the language that are effectively first-order. Overcoming this shortcoming by using existential and universal quantifiers to fully apply lambda-bound abstractions does not solve the problem since SMT is incomplete in the presence of quantifiers and may be able to prove validity for some formulas but not equivalent formulas that vary slightly in order, number or complexity. For this reason, queries

with a large number of partially applied functions are not likely to meet with success without the use of a TPS.

The refinement relationship between the swap specification, the general swap, and the numeric swap was proven valid in a high-order theorem prover.[Motty and Norvell, 2010] The approach for doing so is markedly different and less amenable to automation than that of the SMT solver. Logical proofs in TPS are largely about transformation; they work best when it can be demonstrated that the proof at hand is nothing more than a type-instantiation, alpha conversion, specialization or generalization of an existing theorem. Section 8.1 describes a number of macros that were necessary to get the case studies proven successfully in the TPS known as HOL4. Experience using a specific TPS can improve the odds of finding successful strategies but cannot guarantee success.

Finally TPSs do not permit uninterpretted user-defined functions. Functions must be fully interpreted so that the TPS can prove user-declared assertions about them. This requires a much more stringent and complete definition of subroutines and external dependencies upon which an algorithm relies.

8.1 Kananaskis HOL Case-Study

A 20-year stretch of interactive proof assistants released under the acronym HOL88 culminated with a successor HOL4, also known as Kananaskis HOL. The acronym HOL refers to the use of high-order logics which use a type system to support the handling of partially applied functions as objects of enquiry.

HOL's theories are loaded from libraries. One library in particular, known as

'bossLib', provides a suite of basic automated proving tools. A number of other libraries provide type syntaxes which make it possible to extend HOL's native data types to include numbers, strings and lists. Another important library, mesonLib, defines a number of essential automated model elimination algorithms.

8.1.0 Definitions

The following higher-order definitions for refinement, sequential composition, assignment and subs (local substitution of a variable for an expression inside the body of a statement) were introduced into the theorem prover.

DefinitionOfRefinement $LHS \sqsubseteq RHS \triangleq \forall(s : \text{`}a)\ (s' : \text{`}b).RHS\ s\ s' \Rightarrow LHS\ s\ s'$

Assign $assign\ x\ e\ s\ s' \triangleq \forall y.\ if\ x = y\ then\ (s'y) = (es)\ else\ (s'y) = (sy)$

Sequential Composition $sc\ f\ g\ s\ s' \triangleq (\exists\ s^{\cdot}.\ f\ s\ s^{\cdot} \wedge g\ s^{\cdot}\ s')$

Substitution $subs\ f\ x\ e\ s\ s' \triangleq (let\ s'' = \lambda y \cdot if\ x = y\ then\ es\ else\ sy\ in\ f\ s''\ s')$;

8.1.1 Macros

In order to prove the validity of the relationship between the swap specification, the general swap, and the numeric swap the available theories of HOL were used in ways prescribed by a small number of macros.[Motty and Norvell, 2010] These macros consist of:

REP_EVAL_TAC Exhaustively evaluate expressions in a theorem until no further change results

MAKE_IT_SO Given an expression th, if th is a hypothesis of a theorem (i.e. in the assumption list), simplify it to TRUE anywhere it appears in the conclusion

REFINEMENT_TAC and REFINEMENT_RULE Rewrites references to the refinement function with its first-order definition. This converts an expression $LHS \sqsubseteq RHS$ to $\forall s, s' \cdot RHS\ s\ s' \Rightarrow LHS\ s\ s'$

thmAbstractSpecification Converts the higher-order identity between specifications to a first order functional equivalence, i.e. $(\forall\ s\ s'\ \cdot\ f\ s\ s' \Leftrightarrow g\ s\ s') \Leftrightarrow (f = g)$

thmOnePointLemma Proof of the theorem $(\ x = x\ )\ \wedge\ (\ f\ x\ t\ ) \iff f\ x\ t$

thmForwardSubstitution Proof of the theorem $\forall f\ e\ x\ s\ s' \cdot sc\ (\ assign\ x\ e\ )\ f\ s\ s' = (\ let\ s'' = \lambda y.\ if\ x = y\ then\ es\ else\ s\ y\ in\ f\ s''s'\ )$

EvaluateFor Given a refinement expression in the form of $LHS \sqsubseteq RHS$ if the RHS is often reducible to the form:

$$\lambda s, s' \cdot \exists s^{\cdot} \cdots s^{\cdot\cdot} \cdot \bigwedge_I \left\{ \forall v \cdot \left(\begin{array}{l} if\ \text{“}\alpha\text{”} = v\ then\ F_{I\alpha} s, s^{\cdot} \cdots s^{\cdot\cdot}, s' \\ else\ if\ \text{“}\beta\text{”} = v\ then\ F_{I\beta} s, s^{\cdot} \cdots s^{\cdot\cdot}, s' \\ else\ if \cdots \end{array} \right) \right\}$$

The reduced form is always possible to generate provided the state(s) are known when any variable (e.g. “α”) undergoes an assignment. If so, the companion function (e.g. $F_{I\alpha}$) need simply assert equalities for the variable's value for those target states. An example is discussed in the results section.

Once the RHS is in the reduced form, the expression $LHS \sqsubseteq RHS$ is converted to an implication using REFINEMENT_TAC and the antecedent is stripped

(added to the list of assumptions) using an application of the built-in tactic REPEAT STRIP_TAC. This converts the expression to the form:

$$\bigcup_I \left\{ \forall v \cdot \left(\begin{array}{l} \mathit{if}\ \text{“}\alpha\text{”} = v\ \mathit{then}\ F_{I\alpha} s, s^{\cdot} \cdots s^{\cdot\cdot}, s' \\ \mathit{else\ if}\ \text{“}\beta\text{”} = v\ \mathit{then}\ F_{I\beta} s, s^{\cdot} \cdots s^{\cdot\cdot}, s' \\ \mathit{else\ if} \cdots \end{array} \right) \right\} \vdash \mathit{LHS}$$

The EvaluateFor macro will evaluate the RHS for each state variable (e.g. “α”,“β”) and simplify the *LHS* until the theorem has been exhaustively reduced.

8.1.2 Results

The macros defined above were specifically designed and tested in order to prove theories of stepwise refinement of programs. A number of these proofs are discussed here: specifically a proof of the forward substitution law and a verification that the general swap algorithm and the algebraic swap algorithms meet their specification.

The examples shown in this section are written in HOL's syntax. HOL uses standard keyboard symbols for the following logical symbols:

Logical Symbol	Keyboard Character
$\forall$	!
$\exists$	?
λ	\
$\sqsubseteq$	[=.

8.1.2.0 Forward Substitution Law

The Forward Substitution Law is expressed in HOL as follows:

```
(
    (
        \ (s:string->'b) (s':string->'b) . ((s' "m") = (s "n")) /\ ((s' "n") = (s "m↩
↪"))
    )
    [=.
    (
        sc
        (
      sc (assign ("t") (\ (s:string->'b).s "m")) (assign "m" (\ (s:string->'b). s "n↩
↪"))
        )
        (assign "n" (\ (s:string->'b).s "t"))
    )
)
```

Listing 8.0: HOL Goal: General Swap

"!$f\ e\ x\ s\ s'.\ sc\ (assign\ x\ e)\ f\ s\ s' = subs\ f\ x\ e\ s\ s'$ ",

The proof of the forward substitution law in HOL is surprisingly complicated. It requires two rewrite rules and two lemmas be introduced in addition to the definitions already discussed. With the introduction of these rules and lemmas, the proof still requires 10 steps and is partitioned amongst a number of sub-goals. The proof is recreated in Appendix A and serves mainly as empirical evidence of the difficulty involved in partitioning problems so that the rules and background theory are sufficient to reach a proof in a higher-order system.

8.1.2.1 General Swap Algorithm

The aim of the general swap algorithm is to prove the theorems of Listing 8.0 true in HOL. In order to automate the proof in Kananaskis HOL, the right-hand side of the refinement needs to be converted to the form of Listing 8.1. The conver-

```
0 (?s".
      (!y.if "m" = y then s" y = s "n" else s" y = if "t" = y then s "m" else s y)
        /\
      (!y. if "n" = y then s' y = s" "t" else s' y = s" y)
)
```

Listing 8.1: RHS: General Swap

sion is accomplished passing the substitution $(\forall\ s\ s'\ \cdot\ f\ s\ s'\ \Leftrightarrow\ g\ s\ s') \Leftrightarrow (f\ = g)$ (thmAbstractSpecification) to HOL's built in tactic, SUBST_TAC and then applying the custom macro REP_EVAL_TAC. In this form, the built in HOL tactic REPEAT STRIP_TAC is able to discharge the companion functions into the assumption list. The macro EvaluateFor ["t","x","y"] can be used and the proof solved with a call to REP_EVAL_TAC.

8.1.2.2 Algebraic Swap Algorithm

The aim of the algebraic swap algorithm is to prove the theorems of Listing 8.2 true in HOL. In order to automate the proof, the right-hand side of the refinement needs to be converted to the form of Listing 8.3. The proof of the algebraic swap example depends on theorems in the HOL library arithmeticTheory, and the lemma:

lemma = "!(a:num) (b:num). (a + b -(a + b -b)) = (b + a - a)"

The conversion is accomplished passing the substitution $(\forall\ s\ s'\ \cdot\ f\ s\ s'\ \Leftrightarrow\ g\ s\ s') \Leftrightarrow (f\ = g)$ (thmAbstractSpecification) to HOL's built in tactic, SUBST_TAC and then applying the custom macro REP_EVAL_TAC. In this form, the built-in HOL tactic REPEAT STRIP_TAC is able to discharge the companion functions into the assumption list. The macro EvaluateFor ["x","y"] can be used and the proof solved with

```
(
    \ (s:string->num) (s':string->num). ((s' "m") = (s "n")) /\ ((s' "n") = (s "m↩
↪"))
)
[=.
(
    sc
        (
            ( sc
                (assign "m" (\ (s:string->num).((s "m") + (s "n"))))
                (assign "n" (\ (s:string->num). ((s "m") - (s "n"))))
            )
        )
        (
            assign "m" (\ (s:string->num).((s "m") - (s "n")))
        )
)
```

Listing 8.2: HOL Goal: Algebraic Swap

```
(?s".
        !(y :string). if "n" = y then s" y = s "m" + s "n" - s "n"
                  else s" y = if "m" = y then s "m" + s "n" else s y)
    /\
        !(y :string).if "m" = y then s' y = s" "m" - s" "n" else s' y = s" y)
)
```

Listing 8.3: RHS: Algebraic Swap

a call to (PROVE_TAC [LESS_EQ_REFL, LESS_EQ_ADD_SUB, SUB_EQ_0, ADD_0, lemma]).

8.1.3 Observations

While the macros declared here are not enough to solve every problem poseable by SIMPPLE, two observations can be made:

0. Studying problems in the field of formal methods can lead to the identification of generalized forms suitable for use with automated model elimination algorithms; and

1. For some generalized forms, a set of macros can be identified that increase the success of automated model elimination algorithms

The lemmas and tactics required to prove the various forms of refinement were discovered through iterative interactions with the theorem prover. This involved significant interpretation of error messages and results printed by the prover, as well as a study of the available libraries of theorems built into HOL.

8.2 Summary

This chapter describes an implementation plan for evaluating server-side queries by targeting a higher-order theorem prover. The implementation plan is included in the thesis to demonstrate the process of extending the classes implemented under this thesis to applications beyond the scope of the thesis.

Chapter 9

Conclusion

In this thesis, laws of predicative programming were formulated and tested in two state-of-the-art proof systems. We saw that some programs without loops could be transformed automatically thanks to many of the features built into the Scala language. In this section, the key conclusions are reviewed and the need for future work is discussed.

9.0 Results

Past developments and background work completed prior to the commencement of this thesis produced: a browser-based extension that allows users to edit proofs; and, an abstract syntax and type system for SIMPPLE prototyed in Haskell. This past work served as the starting point to this thesis. The main contribution of this thesis is a Scala language implementation that interfaces to third-party provers. These provers verify the logical correctness of the SIMPPLE program as it is constructed line by line. Two classes of provers were considered: first, Satisfiablilty Modulo Theorem (SMT)

solvers; and secondly higher-order Theorem Proving Systems (TPS). The capabilities of the software was demonstrated. The demonstration consisted of:

- parsing of textual documents conforming to a BNF grammar into abstract syntax trees (ASTs).
- conversion of source abstract syntax trees (ASTs) into first-order logic.
- conversion of first-order logic into target ASTs of an SMT solver.
- an implementation plan for targeting a higher-order TPS for solution of equally or more challenging proofs.

In order to complete the work in the time-frame available, a number of simplifications were necessary, specifically:

- a simplified version of the type-system was used, augmented where necessary by adding the ability to predeclare the types of constants used in the examples.
- the only proofs allowed are those in which the conclusion can be reached through continued refinement of the initial tree.
- no explicit support for loops was included in the language.
- examples were limited to those that could be readily presented using familiar theories of Boolean logic and integer arithmetic.

The software developed as part of this thesis is not limited by the choices made above and can be immediately used to test a wide variety of examples. Furthermore, it was shown in this thesis that in the case of the GCD algorithm, proof of termination

can be established by the SMT solver; consequently support for while loops can be added to the language as a sub-derivation of the derivations in which they appear.

Results are mixed for the demonstration of support for higher-order theorem provers. On the one hand, the type systems of higher-order theorem provers are needed for the most complex proofs; on the other-hand the use of SMT solvers when suitable results in almost effortless automated verification in comparison.

On the whole, the software is expected to be of value. For the purpose of teaching stepwise refinement of software, it allows more challenging proofs to be practiced than can easily be verified manually. For the purpose of research into SMT solvers, it applies theories of predicative programming to SIMPPLE programs, and handles conversion to SMTLIB. It uses an adapter interface to support the targetting and testing of other oracles for future enhancement.

It is hoped that further applications of the software will be found, and that it will contribute to the production of implemented specifications that are maintainable, sharable and unmistakably fit for their purpose.

9.1 Future Work

One of the case studies examined looked at different implementations of the swap algorithm. Both implementations refine the specification for a swap algorithm, and each refines the other as long as the elements being swapped are integers. When two programs equate in such a fashion, they are not necessarily interchangeable. Other factors, such as hardware resource utilization, readability, parallel execution and data representation play a role in determining the suitability of a program for a particular

application.

The examples covered in this thesis involve short segments of code. This was necessary in order to present the results in the space available for a master's thesis. The source code itself is not limited in terms of the number of lines of code in a document, nor the number of variables, constants and theorems introduced therein. Empirical evidence suggests that fairly complex queries can be tolerated by the back-end provers, especially in the case when the refinement is valid. When invalid refinements are made, SMT solvers work hard to find a counterexample but frequently time-out or return unknown. Recent improvements to Z3 have been made and are promising for the types of problems presented here. Especially promising is the addition of quantifier elimination tactics.[De Moura and Bjørner, 2011]

The modular, object oriented nature of the source code developed should aid in extension and debugging. Future work is required to target additional oracles and add comprehensive theories beyond that of integer and boolean logic. Incorporation of loops, arrays and parallel operation of threads is also important in today's environment.

The thesis described a method of developing fully verified software, but the software created in the process of the thesis does not follow the methodology. The software methodology used in this thesis described its inputs and outputs with formal grammars and used UML diagrams to describe the object-oriented architecture of the implementation. As an alternative output of the methodology, the possibility that the specifications can be refined into a higher-order formula in the instruction set of model-driven architectures (MDA) instead of computer instructions should also be considered.

The thesis describes a method of programming by specification that applies to imperative languages, but Scala and many modern languages contain functional aspects. It is noted that approaches to stepwise refinement for functional programming exist. "Functional and imperative programming are not really competitors; they can be used together.... Functional programming and imperative programming differ mainly in the notation they use for substitution" [Hehner, 2014]. Alternatively, code which lends itself to an imperative implementation can be tested in isolation.

References

[Barrett *et al.*, 2010] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard – version 2.0. In *Proceedings of the* 8^{th} *International Workshop on Satisfiability Modulo Theories (SMT '10)*, July 2010. Edinburgh, Scotland.

[Chomsky, 1959] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[de Moura and Bjørner, 2008] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[De Moura and Bjørner, 2011] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications, 2011. `http://dl.acm.org/citation.cfm?id=1995394`.

[Dijkstra, 1975] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[d'Silva *et al.*, 2008] V. d'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *Computer-Aided Design*

of Integrated Circuits and Systems, IEEE Transactions on, 27(7):1165–1178, July 2008.

[Goguen *et al.*, 2000] Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing obj. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ*, volume 2 of *Advances in Formal Methods*, pages 3–167. Springer US, 2000.

[Gries and Schneider, 1993] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math.* Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[Hawking, 2005] S. W. Hawking, editor. *God Created the Integers : the Mathematical Breakthroughs that Changed History.* Running Press, Philadelphia, PA ; London :, 2005.

[Hehner, 1993] Eric C. R. Hehner. *A Practical Theory of Programming.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[Hehner, 2014] Eric C. R. Hehner. A Practical Theory of Programming, 2014. On the web at `http://www.cs.toronto.edu/~hehner/aPToP/aPToP.pdf`.

[Hoare, 1969] C. A. R. Hoare. *An Axiomatic Basis for Computer Programming*, volume 12. ACM, New York, NY, USA, October 1969.

[Morgan, 1994] Carroll C. Morgan. *Programming from Specifications, 2nd Edition.* Prentice Hall International series in computer science. Prentice Hall, 1994.

[Morgan, 2009] Carroll Morgan. How to brew-up a refinement ordering. *Electron. Notes Theor. Comput. Sci.*, 259:123–141, December 2009.

[Motty and Norvell, 2010] Stephen Motty and Theodore Norvell. Interactive proofs of programs. *NECEC 2010*, 2010.

[Norvell, 2009] Theodore S. Norvell. The simple report, 2009. Draft: Typeset October 13. Available from author upon request.

[Norvell, 2012] Theodore S. Norvell. Theory of computing/computation, 2012. Typeset January 5. Available from author upon request.

[Odersky, 2014] Martin Odersky. What is scala, June 2014. On the web at `http://www.scala-lang.org/what-is-scala.html`.

[Reilly *et al.*, 2000] Edwin D. Reilly, Anthony Ralston, and David Hemmendinger. *Encyclopedia of Computer Science*. Nature Publishing Group, London, 2000.

[Research, 2014] Microsoft Research. Z3 guide, June 2014. On the web at `http://rise4fun.com/z3/tutorialcontent/guide#h22`.

[Subramaniam, 2009] Venkat Subramaniam. *Programming Scala: Tackle Multi-Core Complexity on the JVM*. The pragmatic programmers. Pragmatic Bookshelf, Raleigh, NC, 2009.

[Wang, 1961] Hao Wang. The calculus of partial predicates and its extension to set theory i. *Mathematical Logic Quarterly*, 7(17-18):283–288, 1961.

Appendix A

Proof of Forward Substitution Theorem

This chapter provides a code listing of the forward substitution law as proven in the Kananaskis 9 HOL theorem prover.

A.0 Forward Substitution Law

The forward substitution law can be expressed in higher-order logic with the support of lambda expressions as:

$$\forall F\ e\ x\ s\ s\prime.\ \langle assign\ x\ :=e;\ F\ \rangle\ s\ s\prime\ =$$
$$(let\ s" = \lambda y \cdot if\ x = y\ then\ es\ else\ sy\ in\ \langle F\rangle\ s"\ s\prime)$$

A.1 Kananaskis Code Listing

```
set_trace "Unicode" 0;

show_types:=true;
show_assums:=true;

set_fixity "[=." (Infixl 500);
val DefinitionOfRefinement = xDefine "bRefinement"
    ‘v [=. u = !(s:’a) (s’:’b). u s s’ ==> v s s’‘
;

val REFINEMENT_TAC =
    (*
        [
        ] |-
                ‘\ s s’ .v s s’ [=. u
    *)
    (PURE_ONCE_REWRITE_TAC [DefinitionOfRefinement])
                (* [
                   ] |-
                          !s s’. u s s’ ==> \ s s’. v s s’
                *)
    THEN
        (REPEAT GEN_TAC)
                (* [
                   ] |-
                        u s s’ ==> (\s s’. v s s’) s s’
                *)
    THEN
        (BETA_TAC)
                (* [
                   ] |-
                        u s s’ ==> v s s’
                *)
;

fun REFINEMENT_RULE th =
    (
        BETA_RULE
        (
            GEN_ALL
            (
                PURE_ONCE_REWRITE_RULE [DefinitionOfRefinement] th
            )
        )
    )
;

Define ‘assign x e s s’ =
```

```
            !y.
                if x = y then
                    (s' y) = (e s)
                else
                    (s' y) = (s y)
        '
;

Define 'sc f g s s' = (? s'' . f s s'' /\ g s'' s' ) ' ;

fun SWAPLR_RULE th =(PURE_ONCE_REWRITE_RULE [EQ_SYM_EQ] th);

val thmAbstractFunction =
    prove
    (
        ''!(t :'a -> 'b) (f :'a -> 'b).(t = (\(y :'a). f y)) <=> (!(y:'a).(t (y :'a) =
 f y)) '',
        (
            (EVAL_TAC)
                (*  [
                ] |-
                        !t f. (t = f) <=> !y. t y = f y
                *)
        THEN
            (REPEAT STRIP_TAC)
                (*  [
                ] |-
                         (t = f) <=> !y. t y = f y
                *)
        THEN
            (ACCEPT_TAC (SPECL [''t:'a->'b'',''f:'a->'b''] FUN_EQ_THM))
                (* [] |- !t f. (t = (\y. f y)) <=> !y. t y = f y : thm *)
        )
    )
;

val thmConditionalFunction =
    let val
        goal = ''!(t :'a -> 'b) (a :'a -> bool) (b :'a -> 'b) (c :'a -> 'b). (!(y :'a
).
                if a y then t y = b y else t y = c y)
                    <=>
                (t = (\(y :'a). if a y then b y else c y))''
    and
        specializedTerm0 = [
            ''t:'a->'b'',
            ''\(y:'a).(if (a:'a->bool) y then (b:'a->'b) y else (c:'a->'b) y)''
        ]
    and
        specializedTerm1 = [
```

```
        ``\rhs.((((t:'a->'b) (y:'a)) = rhs )) ``,
        ``(a:'a -> bool) (y: 'a)``,
        ``((b:'a->'b) (y:'a))``,
        ``((c:'a->'b)(y:'a))``
    ]
in
    prove
    (
        goal,
        (
            (* [
               ] |-
                    !t a b c. (!y. if a y then t y = b y else t y = c y) <=>
                        (t = (\y. if a y then b y else c y))
            *)
            (REPEAT STRIP_TAC)
                (*  [
                   ] |-
                        (!y. if a y then t y = b y else t y = c y) <=>
                            (t = (\y. if a y then b y else c y))
                *)
            THEN
                (EQ_TAC THENL
                [(
                    (*  [
                       ] |-
                            (!y. if a y then t y = b y else t y = c y)
                                ==> (t = (\y. if a y then b y else c y))
                    *)
                    (REPEAT STRIP_TAC)
                        (*  [
                                (!y. if a y then t y = b y else t y = c y)
                           ] |-
                                (t = (\y. if a y then b y else c y))
                        *)
                    THEN
                        (SUBST_TAC [(BETA_RULE (SPECL specializedTerm0
thmAbstractFunction))])
                            (*  [
                                    (!y. if a y then t y = b y else t y = c y)
                                ] |-
                                    !y. t y = if a y then b y else c y
                            *)
                    THEN
                        (REPEAT STRIP_TAC)
                            (*  [
                                    (!y. if a y then t y = b y else t y = c y)
                                ] |-
                                    t y = if a y then b y else c y       *)
                    THEN
                        (SUBST_TAC [
```

```
                    (BETA_RULE(SPECL specializedTerm1 (
                        INST_TYPE [
                            alpha |-> ``:'b ``, beta |->``:bool``
150                     ] COND_RAND))
                    )
                ])
                    (* [
                        (!y. if a y then t y = b y else t y = c y)
                     ] |-
155                         if a y then t y = b y else t y = c y
                    *)
                THEN
                    (FIRST_ASSUM (ACCEPT_TAC o (SPEC ``y:'a``)))
160             ),(
                    (* [
                    ] |-
                            (t = (\y. if a y then b y else c y))
                                ==> (!y. if a y then t y = b y else t y = c y)
165                 *)
                    (REPEAT STRIP_TAC)
                    (* [
                            t = (\y. if a y then b y else c y)
                        ] |-
170                         if a y then t y = b y else t y = c y
                    *)
                THEN
                    (SUBST_TAC [(
                        SWAPLR_RULE
                            (
175                             BETA_RULE
                                (
                                    SPECL [
                                        ``\rhs.((((t:'a->'b) (y:'a)) = rhs )) ``,
                                        ``(a:'a -> bool) (y: 'a)``,``((b:'a->'b) (y
180 :'a))``,
                                        ``((c:'a->'b)(y:'a))``
                                    ] (INST_TYPE [
                                        alpha |-> ``:'b ``, beta |->``:bool``
                                    ] COND_RAND)
185                             )
                            )
                    )]
                )
                    (* [
190                     t = (\y. if a y then b y else c y)
                        ] |-
                            t y = if a y then b y else c y
                    *)
                THEN
195                 (ASSUME_TAC
                        (
```

```
                                UNDISCH (#1(EQ_IMP_RULE
                                        (
                                            BETA_RULE
                                            (
                                                SPECL
                                                [ ``t:'a->'b``,
                                                    ``\ (y:'a).(if (a:'a->bool) y then (b
:'a->'b) y
                                                                                else (c:'a->'b) y)
``
                                                ] thmAbstractFunction
                                            )
                                        )
                                    ))
                            )
                            (*  [
                                    !y. t y = if a y then b y else c y
                                ,
                                    t = (\y. if a y then b y else c y)
                                ] |-
                                        t y = if a y then b y else c y
                            *)
                        )
                    THEN
                        (FIRST_ASSUM (ACCEPT_TAC o (SPEC ``y:'a``)))
                    )]
                    (*  [
                        ] |-
                                !t a b c. (!y. if a y then t y = b y else t y = c y)
<=>
                                    (t = (\y. if a y then b y else c y)) : thm
                    *)
                )
            )
        )
    end
;

fun EXHAUSTIVELY x =
    (REPEAT (CHANGED_TAC x))
;

val REP_EVAL_TAC =
    (EXHAUSTIVELY EVAL_TAC)
;

val thmAcceptInPlace = UNDISCH (prove (``(v:bool) ==> (v <=> T)``,REP_EVAL_TAC));
val thmRejectInPlace = UNDISCH (prove (``(~(v:bool)) ==> (v <=> F)``,REP_EVAL_TAC))
;

fun USE_CONTEXT (asl:term list) (th:thm) =
```

```
    if (null asl) then th else (UNDISCH (USE_CONTEXT (tl(asl)) th))
;

fun VSUB (v:term) (e:term) (th:thm) =
    USE_CONTEXT (hyp th) (SPEC e (GEN v (DISCH_ALL th)))
;

fun MAKE_IT_SO (th:thm) =
    ((SUBST_TAC [(VSUB ``v:bool`` (concl th) thmAcceptInPlace)]) THEN EVAL_TAC)
;

fun MAKE_IT_NO (th:thm) =
    if(is_neg(concl th)) then
        ((SUBST_TAC [(VSUB ``v:bool`` (dest_neg(concl th)) thmRejectInPlace)]) THEN
 EVAL_TAC)
    else
        ((SUBST_TAC [(VSUB ``v:bool`` (mk_neg(concl th)) thmRejectInPlace)]) THEN
EVAL_TAC)
;

val SPEC_EQ_THM =
    prove
    (
        ``(!(s :'a) (s' :'b).(f :'a -> 'b -> 'c) s s' = (g :'a -> 'b -> 'c) s s') <=>
(f = g)``,
        (
            (EQ_TAC THENL
                [(
                    (*  [
                        ] |-
                                (!s s'. f s s' = g s s') ==> (f = g)
                    *)
                    (DISCH_TAC)
                        (*  [
                                    (!s s'. f s s' = g s s')
                            ] |-
                                    (f = g)
                        *)
                THEN
                    (SUBST_TAC [(INST_TYPE [beta |-> ``:'b->'c``] (SPEC_ALL
FUN_EQ_THM))])
                        (*  [
                                    (!s s'. f s s' = g s s')
                            ] |-
                                    !x. f x = g x
                        *)
                THEN
                    (GEN_TAC)
                        (*  [
                                    (!s s'. f s s' = g s s')
                            ] |-
```

```
                                f x = g x
                        *)
                    THEN
                        (SUBST_TAC [(SPECL [``(f:'a->'b->'c) (x:'a)``,``(g:'a->'b->'c)
    (x:'a)``] (
295                         INST_TYPE [alpha |-> ``:'b``, beta |-> gamma] FUN_EQ_THM
    ))]
                        )
                            (*  [
                                    (!s s'. f s s' = g s s')
                                ] |-
300                                  !x'. f x x' = g x x'
                            *)
                    THEN
                        (GEN_TAC)
                            (*  [
305                                 (!s s'. f s s' = g s s')
                                ] |-
                                     f x x' = g x x'
                            *)
                    THEN
310                     (FIRST_ASSUM (ACCEPT_TAC o (SPECL [``x:'a``,``x':'b``])))
                    ),(
                        (*  [
                            ] |-
                                    (f = g) ==> !s s'. f s s' = g s s'
315                     *)
                        (REPEAT STRIP_TAC)
                            (*  [
                                    (f = g)
                                ] |-
320                                 !s s'. f s s' = g s s'
                            *)
                    THEN
                        (REPEAT AP_THM_TAC)
                            (*  [
325                                 (f = g)
                                ] |-
                                    f = g
                            *)
                    THEN
330                     (FIRST_ASSUM ACCEPT_TAC)
                    )]
                )
            )
                (*
335                 [] |- (!s s'. f s s' = g s s') <=> (f = g) : proof
                *)
        )
    ;
```

```
val thmAbstractSpecification =
    INST_TYPE [
        alpha |-> ``:'a -> 'b``, beta |-> ``:'a -> 'b``, gamma |-> ``:bool``
    ] SPEC_EQ_THM
        (*
            [] |- (!s s'. f s s' <=> g s s') <=> (f = g) : thm
        *)
;

val thmOnePointLemma=
    prove
    (
        `` (x = x) /\ (f x t ) <=> f x t``,
        (
            (EQ_TAC THENL
                [(
                    (*  [
                        ] |-
                                ((x = x) /\ (f x t )) ==> f x t
                    *)
                        (ASSUME_TAC (REFL ``x``))
                        (*  [  (x = x)
                            ] |-
                                    ((x = x) /\ (f x t)) ==> f x t
                        *)
                    THEN
                        (FIRST_ASSUM MAKE_IT_SO)
                ),(
                    (*  [
                        ] |-
                                (f x t) ==> (x = x) /\ f x t
                    *)
                        (DISCH_TAC)
                        (*  [
                                (f x t)
                            ] |-
                                    (x = x) /\ f x t
                        *)
                    THEN
                        (FIRST_ASSUM MAKE_IT_SO)
                )]
            )
        )
    )
;

Define `subs f x e s s'
             = (let s'' = \y. if x=y then e s else s y
                in f s'' s') `;

val thmForwardSubstitution =
```

```
    let val
        conversion0 = BETA_RULE
                        (
                            SWAPLR_RULE
                            (
                                SPECL
                                    [
                                        ‘‘s’’:’a->’b‘‘,
                                        ‘‘\(y:’a).((x:’a) = y)‘‘,
                                        ‘‘\(y:’a).((e:(’a->’b)->’b) (s:’a->’b))‘‘,
                                        ‘‘s:’a->’b‘‘
                                    ]
                                        thmConditionalFunction
                            )
                        )
    and
        conversion1 = SWAPLR_RULE
                        (
                            BETA_RULE
                            (
                                SPECL
                                    [
                                        ‘‘s’’:’a->’b‘‘,
                                        ‘‘\ (y:’a).if (x:’a) = y then (e:(’a->’b)->’b) (s
:’a->’b) else s y‘‘
                                    ]
                                        thmAbstractFunction
                            )
                        )
    and
        lemma0 = BETA_RULE
                        (
                            SPECL
                                [
                                    ‘‘s’’:’a->’b‘‘,
                                    ‘‘\(y:’a).if (x:’a) = y then (e:(’a->’b)->’b) (s:’a
->’b) else s y‘‘
                                ]
                                    thmAbstractFunction
                        )
    and
        lemma1 = VSUB ‘‘t:’c‘‘ ‘‘s’:’c‘‘
                        (
                            VSUB ‘‘x:’a->’b‘‘ ‘‘\(y:’a).if(x = y)  then (e:(’a->’b)->’
b) (s:’a->’b)
                                                    else s y‘‘
                                (INST_TYPE [
                                    alpha |-> ‘‘:(’a->’b)‘‘ , beta |-> ‘‘:’c‘‘
                                ] thmOnePointLemma)
                        )
    in
```

```
prove
(
    ``!f e x s s'. sc (assign x e) f s s' = subs f x e s s' ``,
    (
        (REPEAT STRIP_TAC)
            (*  [
                ] |-
                        sc (assign x e) f s s' <=> subs f x e s s'
            *)
        THEN
            (EVAL_TAC)
            (*  [
                ] |-
                        (?s''.
                            (!y. if x = y then s'' y = e s else s'' y = s y)
/\ f s'' s')
                        <=>
                            f (\y. if x = y then e s else s y) s'
            *)
        THEN
            (REWRITE_TAC [(REWRITE_RULE [conversion0] lemma0),conversion1])
            (*  [
                ] |-
                        (?s''.
                            (s'' = (\y. if x = y then e s else s y)) /\ f s''
s')
                        <=>
                            f (\y. if x = y then e s else s y) s'
            *)
        THEN
            (SUBST_TAC [(SWAPLR_RULE lemma1)])
            (*  [
                ] |-
                        (?s''.
                            (s'' = (\y. if x = y then e s else s y)) /\ f s''
s')
                        <=>
                                ((\y. if x = y then e s else s y) =
                                    (\y. if x = y then e s else s y))
                            /\
                                f (\y. if x = y then e s else s y) s'
            *)
        THEN
            (EQ_TAC THENL
            [(
                (*  [
                    ] |-
                            (?s''.
                                (s'' = (\y. if x = y then e s else s y)) /\ f
s'' s')
                            ==>
```

```
                                        ((\y. if x = y then e s else s y) =
                                            (\y. if x = y then e s else s y))
                                    /\
                                        f (\y. if x = y then e s else s y) s'
490
                        *)
                        (DISCH_TAC)
                            (*  [
                                    (?s''. (s'' = (\y. if x = y then e s else s y)) /\
    f s'' s')
495                             ] |-
                                        ((\y. if x = y then e s else s y) =
                                            (\y. if x = y then e s else s y))
                                    /\
                                        f (\y. if x = y then e s else s y) s'
500                         *)
                    THEN
                        (FIRST_ASSUM CHOOSE_TAC)
                            (*  [
                                    (s'' = (\y. if x = y then e s else s y)) /\ f s''
    s'
505                             ,
                                    (?s''. (s'' = (\y. if x = y then e s else s y)) /\
    f s'' s')
                                ] |-
                                        ((\y. if x = y then e s else s y) =
                                            (\y. if x = y then e s else s y))
510                                  /\
                                        f (\y. if x = y then e s else s y) s'
                            *)
                    THEN
                        (FIRST_ASSUM (fn th => (TRY(REWRITE_TAC [(SWAPLR_RULE
    th)]))))
515                 ),(
                        (*  [
                            ] |-
                                    ((\y. if x = y then e s else s y) = (\y. if x = y
    then e s else s y))
                                        /\
520                                 f (\y. if x = y then e s else s y) s'
                                    ==>
                                    (?s''. (s'' = (\y. if x = y then e s else s y)) /\
    f s'' s')
                        *)
                        (DISCH_TAC)
525                         (*  [
                                    ((\y. if x = y then e s else s y) = (\y. if x = y
    then e s else s y))
                                        /\
                                    f (\y. if x = y then e s else s y) s'
                                ] |-
```

```
                                    (?s''. (s'' = (\y. if x = y then e s else s y))
    /\ f s'' s')
                            *)
                        THEN
                            (FIRST_ASSUM (fn th => (TRY (
                                EXISTS_TAC ((#1(dest_eq(#1(dest_conj(concl th))))))))
                            )
                            )
                                (*  [
                                        ((\y. if x = y then e s else s y) = (\y. if x = y
    then e s else s y))
                                            /\
                                        f (\y. if x = y then e s else s y) s'
                                    ] |-
                                            ((\y. if x = y then e s else s y) = (\y. if
     x = y then e s else s y))
                                            /\
                                            f (\y. if x = y then e s else s y) s'
                            *)
                        THEN
                            (FIRST_ASSUM MAKE_IT_SO)
                        )]
                    )
                )
            )
        end
    ;
```

Listing A.0: Proof of Forward Substitution Law