ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Verification of Imperative Programs in Scala

Régis Blanc

School of Computer and Communication Sciences
Master's Thesis

Supervisors:
Professor Viktor Kuncak
Philippe Suter

June 22, 2012

Abstract

Safety-critical software systems can only support a limited number of failures. Extensive testing is good at catching errors, however that will never certify their absence. Formal verification is an alternative to testing that can (automatically) provide a mathematical proof of correctness of programs. In this thesis, we present a verification procedure for imperative programs. Our procedure reduces imperative programming to functional programming and uses a semi-decision procedure that can reason modulo recursive functions. As a complementary method, we propose an algorithm to generate test cases that attain a high coverage of the program statements or can force the execution of some very refined control paths. We have implemented these algorithms and have integrated them in the Leon verification system. Leon can be used to verify programs written in a proper subset of Scala.

Acknowledgements

This thesis has been written during a four-month long project which would definitely not have been possible without the help of many people. I would first and foremost like to thank my supervisor, Viktor Kuncak, whose continued enthusiasm and excellent guidance have driven this work from beginning to end. I am also very grateful that he has given me the opportunity to work on such challenging topics. I would like to express my thanks to Philippe Suter for his constant feedback as well as his great advices on how to tackle the many problems I faced during this thesis. I also appreciate his patience while I was adapting to the Leon system and its internal working. I would also like to thank all the members of the Laboratory for Automated Reasoning and Analysis for the great ambiance during these four months, and in particular Etienne for some fruitful discussions. I am also thankful towards Diggory Hardy and Marc Zimmermann who, through their class project, have helped me identify several issues with the system I was developing and gave me the possibility to address them before the end. Finally, a huge thanks to all the anonymous readers and reviewers of this document.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Writing good and reliable software is a difficult art. Software tend to contain bugs, and, unfortunately, bugs can have expensive consequences. Basic software engineering principles suggest a number of specific guidelines to follow. These guidelines are certainly likely to reduce the density of errors in a program, however they cannot guarantee the total correctness of the software.

The most common approach to reduce the number of faults is to manually test the program under development. Some values are selected and used as inputs for the program, then some properties of the system, or its output, are asserted and dynamically checked. This process can be automated to some extend: the collection of tests can be recorded and automatically run and checked each time the program is modified. This certainly improves productivity, but this does not replace the need to painfully compose a good set of test values.

In addition, one single test can only exercise one path in the control flow of the program. But the number of different paths of executions a program can take is gigantic. Indeed, for each occurrence of a conditional expression with k branches, the number of possible paths of executions is multiplied by k. If we simply consider a program containing n `if-then-else` statements, which have $k = 2$ branches, there are already 2^n possible paths: an exponential growth. Actually, the complete story is even worse: in the presence of loops or recursive functions, there is an infinite number of paths. Not to mention that a total coverage of all paths of executions, if at all possible, would not ensure the absence of bugs. It is possible for distinct values to lead to the same execution through the control flow.

Nevertheless, testing is still very important. In the absence of more formal methods, it is mandatory to have a decent test suite that can at least give some confidence in the correctness of the software. Testing is also very cost effective compared to many formal methods that usually involve massive user assistance. This last point is slightly less relevant when the testing is done manually, forcing the user to write good test cases that have good code coverage.

Since testing can never prove the correctness of the software and can at best exhibits erroneous behaviour. We need to turn to formal verification if we want stronger guarantees. Formal verification can provide a mathematical proof that a program meets its specification. Such a proof would ensure, assuming the specifications correctly capture the intended behaviour, that the software absolutely cannot fail to complete its task.

The general question of determining if a program meets its specification is undecidable. So, to tackle this problem, one has either to identify decidable subproblems, or accept to rely on semi-decision procedures and heuristic methods.

In this work, we focus on software verification of imperative programs. We leverage an existing semi-decision procedure for verification of purely functional programs with recursive functions [59]. We apply a series of successive transformation passes to the input program, each pass eliminating some imperative features by mapping them to an equivalent functional feature. In the end, the input imperative program is entirely transformed to a functional program, and we can fall back on the procedure we mentioned above.

Our verification system has the property to be complete for finding counter-examples. It is a semi-decision procedure because it can loop forever on some valid instances, but it is still able to sometimes provide a proof of correctness. However, in practice, we often need inductive reasoning to prove complex properties. To take advantage of the efficient detection of bugs, we advocate the use of our system to actively check properties while developing the software. If the tool finds a counter-example, then the developer obtains a concrete input on which the property is not true, hence he can hopefully address the issue. If the tool is not able to quickly find a counter-example, one should assume that the property holds and continue with the development.

One persistent critic targeting formal verification is that the use of any of its tools requires to be an expert and that mere programmers will never be able to do verification on their own. For example, writing specifications usually requires knowledge about first order logic, a skill that average programmers may lack. The need to learn an additional syntax can also be problematic. In our system, the specification language is actually identical to the implementation language, using existing facilities of the host language to provide specifications. In that setting, writing specifications feels natural, and we think this is an important step towards encouraging the wide adoption of formal verification.

As a complementary method, we extend our system to automatically generate test cases for the program under verification. Automatic test cases generation has the cheap cost of manual testing without having its need for extensive user participation. If the verifier times out, this likely means that the property is valid but the system is not able to prove it so. At that point, we can attempt to generate a collection of test cases that provide a high coverage of program statements. Our technique is actually able to guarantee a complete coverage of all program statements that are reachable starting from an annotated function.

1.1 Motivation

Program verification has a lot of practical and important applications. In a perfect world, any system would be formally verified. In a close to perfect world, any life-critical system would be formally verified, since, in such systems, a single bug can not only cause damage worth billions of dollars but also take some invaluable human lives. Unfortunately, we are not living in either of these worlds, and formal verification of complex systems is very difficult. When a system is too difficult to be completely verified, testing is a good and acceptable alternative.

It is sadly true that the process of formally verifying software can be very expensive in developer time. This usually requires to write down specifications in some formal language like first order logic. Those specifications can often be tricky to get right, and can be an additional source of mistakes. Sometimes the verifier needs hints in order to complete the proof, and, usually, only experts are able to annotate the program exactly the way it should be.

In some applications, errors are more acceptable than in other ones. For such applications, software verification can sometimes reveal itself to be too expensive to compensate for the added value of having lower risk of errors. This is a place where completely automatic tools, like test cases generators, can shine.

Automatic generation of test cases is a very useful method to help in the development of software. Of course, it is neither complete for proving validity or finding counter-example, but it provides a very cheap way to obtain a collection of input values that can exercise the program and give good confidence in its correctness. This is cheap since it does not need any formal specification to be given by the programmer, it can work out of the box on any program.

1.2 Contributions

This thesis makes the following contributions:

1. We present a method to convert an imperative program into an equivalent functional one. We describe the transformation formally using rewrite rules.

2. We show how our transformations can be used to build a verification system for programs written with a mixture of imperative and functional features.

3. We demonstrate how similar methods can be used to automatically generate test cases with a high coverage of program statements.

4. We have implemented the algorithm presented in this thesis in a robust and usable system. The system works on a well-defined subset of Scala. It will clearly flag unsupported constructs Our system can thus be used to develop and check programs as long as they are restricted to the accepted language.

1.3 Outline

The structure of the thesis is as follows. In Chapter 2, we present some important existing techniques in program verification and decision procedures. Our work takes advantage of the availability of these methods. Then, in Chapter 3, we present a high level view of our approach to the verification of imperative programs. In the following two chapters, we delve into the specifics of our method: Chapter 4 explores the different transformation phases that we apply to the input program and Chapter 5 explains how to automatically generate test cases with maximum coverage of statements.

We detail the integration in the existing Leon system in Chapter 6. Chapter 7 then describes how the system behaves on a collection of selected test cases. Finally, Chapter 8 concludes with a discussion of the results and lessons learned from this work.

Chapter 2

Background

In this chapter, we present some existing results that we build upon. First we look at theorem provers and their important applications. Theorem provers are the mandatory building blocks on top of which we can construct verification systems. Then, we will discuss the general problem of software verification. These sections are intended as a high-level introduction to those important topics and not a review of advanced literature. Finally, in the last section, we cover literature on related work.

2.1 Theorem Provers

Theorem provers are specialized software that automate mathematical proofs. They can be entirely automatic, or only partially in the case of interactive theorem provers. Depending on the underlying logic, the problem can vary from easy to undecidable.

In general, proving formula of First-Order Logic (FOL) is undecidable. Despite this issue, FOL is very expressive and, thus, is often used as the specification language for software verification. One pragmatic approach is taken by interactive theorem provers. Here, only some parts of the proof are automated, while the user is required to provide indications for the other parts.

An alternative way to work around the undecidability is to identify decidable fragments that are of interest. Perhaps the most studied such provers are SAT solvers, that only consider propositional logic. SAT is one of the quintessential NP-complete problems. Satisfiability Modulo Theories (SMT) is a generalization of SAT to other theories. There also exists some fully automated first-order theorem provers, which attempt to solve arbitrary first-order logic formula, but they only offer a semi-decision procedure.

2.1.1 Satisfiability Modulo Theories

The SMT problem is the generalization of the SAT problem to any mathematical theory. It has great applications in software analysis, where one usually needs to be able to reason at the word-level: integers, booleans, and data structures.

Specialized theories to answer questions about most of the different fundamental types of objects programming languages use have been developed for many years. A decision procedure for a theory usually takes advantage of some particular knowledge about the theory, to quickly determine the validity of the input formula.

Despite the absence of quantifiers, the expressive power of SMT solvers remains high enough for a lot of practical applications. First note that formulas such as:

$$\exists \overrightarrow{x}.F$$

can be solved by asking whether F is satisfiable. Similarly the following kind of formulas:

$$\forall \overrightarrow{x}.F$$

is equivalent to:

$$\neg \exists \overrightarrow{x} \neg F$$

which can be solved by asking whether $\neg F$ is unsatisfiable. Verification conditions extracted from programs are generally of a similar form and can be solved using an SMT solver. In conclusion, SMT solvers work as long as there are no quantifier alternations.

Satisfiability Modulo Recursive Function

One theory of interest is the one of recursive functions. Recursive functions are expressive enough to give full Turing-completeness power to a programming language, without the need for loops. Most functional programs rely heavily on recursive functions and make only minor to no use of classical imperative looping constructs.

There exists a semi-decision procedure for the theory of recursive function [59]. We only give a high level overview of the algorithm, we refer the reader to the original presentation for additional details.

The procedure is based on successive unrolling of the definition of functions, adding each time more information about how the function behaves. A top level loop alternate over-approximation and under-approximation of the formula, asking an underlying solver each time, until it converges to a solution (or it loops forever).

In the over-approximation phase, all function invocations appearing in the formula are replaced by uninterpreted functions and then the complete formula is sent to a solver. Since the solver actually has the freedom to assign any meaning to the functions, if it still returns that the formula is unsatisfiable, then the actual formula is really unsatisfiable. On the other hand, if the solver says that the formula is satisfiable, we cannot be sure that the meaning chosen for the functions is consistent with their definitions, so we need to continue with the under-approximation phase.

In the under-approximation phase, the formula is crafted in such a way that the model is forced to take only branches that correspond to terminal cases of recursive functions. Such branches always exist, unless the recursive function is non terminating. This means that if this modified function is satisfiable, then the actual function is satisfiable as well. However, if the solver says it is unsatisfiable, we cannot conclude anything about the formula.

2.2 Program Verification

Program verification aims at formally proving the correctness of a program. In general, one can only define the correctness of a program with respect to some specifications. So, given a specification, a software is said to be *partially correct* if, for each terminating execution, the specification are met. We say that a software is *(totally) correct* if it always terminates and the specifications are always met.

The goal is to provide mathematical evidences that an input program meets its specifications. In this work, we will assume that the specifications are given by an external source, usually the programmer. Note that some techniques are known to automatically discover some of these specifications [41, 25].

Figure 2.1 illustrates the general settings of software verification. As we can see, a verification system needs two distinct inputs: the program and the specifications that the program should meet. In practice, it is usually the case that both are combined in a

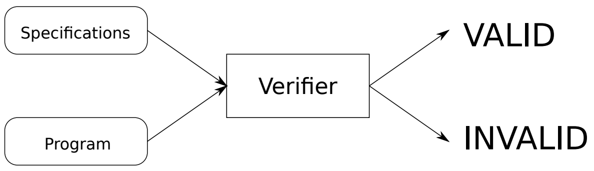

Figure 2.1: Software verification.

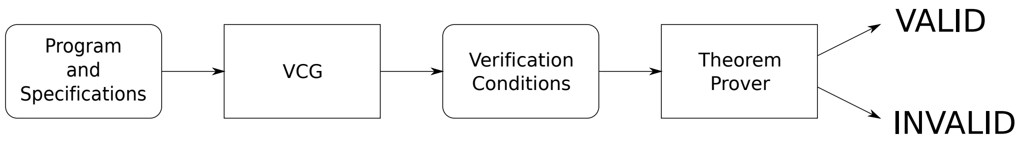

Figure 2.2: Verification conditions.

single input, for example with specifications annotated directly in the source code of the program.

The verification system should then output `VALID` if the program as defined in the source code meets the formal specifications. It can optionally give a proof of the correctness. If the program does not satisfy the specifications, then the verifier should output `INVALID` and optionally give a concrete model (a counter-example). Since program verification is an undecidable problem, it is also possible that the system loops forever, or, similarly, ends up with a timeout and is not able to answer the question.

We can represent programs as logical formulas, and use automatic theorem provers to attempt to solve the formulas. Figure 2.2 shows this process. The input to the system is still the program along with its specifications. In a first phase, the verification conditions generator (VCG) generates formula whose validity implies that the program meets the specifications. The generated formulas are called verification conditions (VC).

The second phase involves calling an external theorem prover to determine the validity of the verification conditions, which permits to conclude the correctness of the program. If a formula is not valid, this means that the program does not meet the given specifications. There could be two reasons for that:

- The program contains a bug. This is the expected behaviour, the specifications correctly capture what the program is supposed to do, and an error in the code leads to a counter-example in the verification conditions.
- The specifications are not correct. In that case, the program might still be correct, but since the specifications wrongly capture the intended behaviours of the program, a counter-example is found.

2.3 Related Work

Many interactive systems that mix the concept of computable functions with logic reasoning have been developed, ACL2 [30, 29] being one of the historical leader. Such systems have practical applications in industrial hardware and software verification [46]. ACL2 is a system that is programmable in a first-order functional subset of Common Lisp [58]. Similarly to our system, it is possible to state theorems using functions returning boolean values. The solving technology is fundamentally very different from ours. ACL2 is based on heuristics with a rewriting system, while we rely on SMT solving technology. The system requires manual assistance because it is usually required to break down a theorem into many small lemmas that are individually proven. Other more recent systems for functional programming include VeriFun [61] and AProVE [22]. Isabelle [51] and Coq [11] are proof assistant systems that provide a programming language in higher order logic to express theorem and help proving them. This logic is expressive enough to define some computable functions in a similar way as it would be done in functional programming. It is actually possible to automatically generate code for such systems [26].

A common trait to these systems is that the outcome is relatively difficult to predict. These systems provide very expressive input languages that make it very hard to automatically solved in general. Many of these systems are also very good at automating the proof of some valid properties, mostly by a smart usage of induction, while our system is complete for finding counter-examples. We think that our approach is more suited for practical programmers, that may not be verification expert but that would be able to make sense out of counter-example, in the form of inputs to the program, much more easily than of a proof of correctness. Also, our system proved to be very capable at finding counter-examples with minimal user guidance. Also, none of these systems are able to reason about imperative code. They are built around a functional core, which naturally mapped to the logic use for formal reasoning.

Our back-end solver can be seen as a form of SMT solver [8]. SMT solvers are complete decision procedures for some class of formulas. Our solver behaves as a semi-decision procedure for formulas containing recursive functions. In the absence of such functions, it will be as complete as the underlying solver. It is an important distinction from other SMT solvers that usually only consider uninterpreted functions or functions interpreted by some fixed axioms of the theory, but never user-defined functions. It is possible to introduce such definitions in an artificial way using universal quantifier. In particular, the function:

$$f(x) = e$$

where e is an expression depending on x, can be defined in a formula as:

$$\forall x. f(x) = e \ .$$

This can then be added as an additional clause for the formula to verify. However, the introduction of universal quantifiers makes the underlying decision procedure incomplete and gives unexpected results. Leon relies on the state-of-the-art solver Z3 [17]. Other well-maintained solvers include CVC3 [10], MATHSAT [13], and YICES [19].

CBMC [15] uses Bounded Model Checking to prove some safety properties of low level C programs. CBMC uses incremental loop unrolling which is in some sense the same idea as our function unrolling method. From a user perspective, CBMC is almost totally automated, while Leon requires the user to write down some specifications. On the other hand, CBMC can only prove some safety properties such as the absence of null dereferences, while Leon can be used for much more general correctness properties. Several tools exist for the verification of contracts and invariants in imperative programs. One such tool is Dafny [38]. Dafny supports an imperative language as well as many object-oriented features. It is thus able to reason about class invariant and mutable fields, which Leon does not support so far. Dafny translates its input program to an intermediate language, Boogie [7], from which verifications conditions are then generated. The generation of verification conditions is done via the standard weakest precondition semantics [18, 50].

Our approach, on the other hand, translates the imperative code into functional code and does not make use of predicate transformers.

Other tools for imperative and object oriented programming include the LOOP verifier [60] and the KEY tool [1] that can verify Java programs annotated by the JML specification language [37]. These tools are less automated than Leon because they rely on an interactive theorem prover instead of an SMT solver as Leon does. The JML specification language offers a rich way to annotate Java program, but it uses a distinct syntax from Java and has not the advantage of being well integrated with the real compiler like Leon. Jahob [63] is another system for Java that can verify full functional correctness for a class of linked data structures. It provides a specification language that supports higher order logic in order to precisely capture the exact behaviour that these data structures should meet.

Some programming languages are also designed with verification in mind. Such programming languages usually have built-in features to express specifications that can be verified automatically by the compiler itself. Such languages include Spec# [6], Gypsy [3] and Euclid [35]. Additionally, Guardol [27] is a domain-specific language designed to build safe and correct network guards. It also uses at its core a translation from imperative to functional and rely on an independent implementation of the same method to solve the resulting verification conditions. However, the language does not support loops and thus do not deal with their translations. Finally, Eiffel [44] is a pioneer in introducing the concept of design by contract, in the form of preconditions and postconditions of functions as language annotations. Eiffel initially only provided, as part of the language, dynamic support for verifying such contracts and in that sense is similar to the contract library for Scala [53].

Chapter 3

Verification of Imperative Programs

In this chapter, we show how to encode an imperative program into verification conditions that imply the correctness of the program, and then call an efficient solver to prove or disprove the conditions. As discussed in the previous chapter, there exists a semi-decision procedure that can solve any decidable theory extended with recursive functions. This allows us to consider the function invocations as part of our logic. This means that we do not abstract a function call by the contract of the function, and thus are very precise by considering the exact definition of the function.

3.1 Design by Contract

Design by contract [45] is a formal approach to software engineering. It has its root in formal verification and Hoare logic. The idea is that a software component should have a very precise interface, expressed by its contract. Each contract can then be individually tested or automatically verified. Even in the absence of formal verification, this is a good practice that is likely to produce modular software.

A contract is made of three parts:

Precondition: what must be true when entering the component.

Invariant: what remains always true.

Postcondition: what is guarantee to be true when leaving the component.

In practice, a component can be as simple as a function that is annotated with a contract. If the function is *pure* (it has no global side-effects), then there is no need for invariant. The contract is then only made of two parts:

Precondition: a boolean expression of the function parameters.

Postcondition: a boolean expression of the function parameters and the returned value.

The goal of our verification phase is then to check the following two properties for each function:

1. If the precondition holds, then the postcondition holds. This property alone is sufficient to prove that the function is correctly implemented.
2. All the invocation of the functions must respect the precondition. If one such condition is violated, then the blame is on the call site and not the function whose precondition is not respected.

```
program    ::= definition*

definition ::= "def" id "(" idlist ")" "=" expr
definition ::= "abstract def" id "(" idlist ")"

expr       ::= lit
expr       ::= "if" "(" expr ")" expr "else" expr
expr       ::= "val" id "=" expr "in" expr
expr       ::= expr "(" exprlist ")"
expr       ::= "(" exprlist ")

lit        ::= INTLIT | BOOLLIT | TUPLELIT
exprlist   ::= _ | (expr ",")* expr
idlist     ::= _ | (id ",")* id

TUPLELIT   ::= "(" ( _ | (lit ",")* lit ) ")"
```

Figure 3.1: Abstract syntax for the purely functional core.

```
expr ::= expr* expr
expr ::= "while" "(" expr ")" expr
expr ::= "var" id "=" expr "in" expr
expr ::= id "=" expr
expr ::= "def" id "(" idlist ")" "=" expr "in" expr
expr ::= "abstract def" id "(" idlist ")" "in" expr
expr ::= "epsilon" "(" id "=>" expr ")"
expr ::= "skip"
```

Figure 3.2: Abstract syntax for extensions.

3.2 The Input Language

Later, we will discuss a number of algorithms manipulating code. In order to keep the presentation as simple as possible, we define an abstract syntax for a simple programming language. The language is minimal, but still Turing-complete. Despite its minimality, it still exhibits all of the important features that need special care in the various manipulations. In our actual implementation we handle real Scala code, which is a bit richer than our input language presented here. In particular, we also handle case classes and pattern matching. Additionally we handle more data types including some data-structure. The algorithms we present with this simplified language are straightforward to generalize.

3.2.1 Syntax

Figures 3.1 and 3.2 describe the complete abstract syntax. The `*` represents number of repetition of the rule, while keywords are between `""`. A `|` marks a disjunction while `_` is used for matching the empty string. Finally, `()` are used to group related items. The rule for `id` is not given but should be assumed to be a standard identifier. `BOOLLIT` is either `true` or `false` while `INTLIT` is any integer number.

Figure 3.1 shows what we will consider as the core language. This core is side-effect free and, since it supports recursive functions, Turing-complete. Figure 3.2 presents a number of extensions to the core language, in particular it adds imperative features with assignments, loops, and blocks of expressions (also called sequencing operator). The sequencing operator is not properly an imperative feature, but it is of no use when there is no expression with side effect. It also provides an `epsilon` construct that allow some form of constraints programming. We also integrate tuples as part of the core language. They will be needed to encode imperative features into functional code.

For the readability of our example code fragments, we will often use arithmetic on numbers. Expression such as `x + y` are not formally part of our grammar, however we will use them as some sort of syntactic sugar for `+(x, y)` which would behaves as a function call of the function with id `+`. We will also assume that all of the standard arithmetic and boolean operations are available. Tuple selectors are also usable, using the syntax `._N` with $N > 0$. They can be considered as a function with one argument for all of our code manipulations.

Note that this grammar is actually ambiguous and we would need a special syntax to mark the end of if, `while`, and definitions expressions. This might look redundant, but it is actually necessary to unambiguously distinguish different expressions. Consider as an example the following expression:

```
if(b)
  x = x + 1
else
  x = x - 1
  y = y + 1
```

Obviously the layout suggest that `y = y + 1` is only executed in the else branch, however it could be also parsed as:

```
if(b)
  x = x + 1
else
  x = x - 1
y = y + 1
```

Here the `y = y + 1` is executed each time at the exit of the `if` expression. Adding these markers for the end of `if` and `while` expressions resolve the this ambiguous behaviour. However this would make the syntax heavy to read so we will rely on the layout and the context to make it clear what parsing is intended. Similar ambiguous parsing can occur with the scope introduced by `var`, here again we will use indentation to make it clear where the scope should end.

Finally, we represent the specifications as code annotations. Top level functions as well as nested functions can be preceded by `@prec expr` and `@post expr` to specify their contracts, and `while` can be preceded by `@inv expr` to specify a loop invariant. As a simplification for the rest of this thesis, we will assume that if no pre/postconditions or invariants are specified, then the default specification of `true` is assumed. Finally, the language supports the notion of abstract functions. Those are functions without implementation, usually annotated with at least a postcondition, and they are assumed to terminate and meeting their contract. In some sense, this is a programming style that is enabled by program synthesis [42, 24].

3.2.2 Semantics

The semantic of this language is as expected. Note that `val`, `var` and `def` introduce a new scope with an identifier that is only visible in the expression after the `in` keyword, except for the function introduced by the `def` construct that needs to be visible in its own body in order to make recursive calls. Additionally, nested functions cannot access mutable variables that are in enclosing scope. This restriction simplifies the code transformations presented in the next chapter.

The language is dynamically typed, and all expressions are integers. We use the convention that `false = 0` and any other value is `true` as is very common in programming languages. For clarity of presentation, we will extensively use `true` and `false` in the rest of this the thesis.

The evaluation order is strict and from left to right. As it is common, `if` expressions only evaluate the branch that is taken by the execution. Since everything is an expression, we will take the convention that assignments and loops returns the value of `false`. The `skip` expression can be used as an empty statement, but is not really mandatory since `false` can be used with a similar effect. Such expressions are useful when we want to use `if` expressions without an `else` case. In fact, we will use the following:

```
if(b)
  e
```

as a syntactic sugar for:

```
if(b)
  e
else
  skip
```

The `epsilon` expressions takes as input an anonymous function, also known as lambda expression. This function should behave as a predicate. The semantic of `epsilon` is then to return any value that would make the predicate `true`, if one such value exists. It will be more thoroughly detailed in Section 4.1.

Abstract functions are used as functions that meet certain contracts but with no implementation provided. It is legal to call an abstract function and the semantics should evaluate the call in a manner consistent with the contract of the abstract function. Also note that we will additionally require that such functions behaves as mathematical, deterministic functions. That is, for the same input values it will return the same output value. This simplifies the mapping to the underlying logic, since in mathematics, all functions are deterministic. Note that we do not explicitly synthesize an equivalent function, but the verification procedure treats it as if it would have this semantics.

The tuple selector `._N` selects the Nth element of the tuple. For example `(5, 7)._2` would evaluate to `7`.

3.3 Generation of Verification Conditions

An imperative programming paradigm usually requires to transform each statement into a relation, in the form of a first order formula, between the program states [28, 18]. By composing such relations for sequences of statements, we end up with a relation for the whole program. This relation can then be used to relate preconditions and postconditions and express the verification conditions.

In the case of functional programming paradigm, since no value changes it is not necessary to represent the program as a relation. Indeed, one can usually use the program expression directly as a term expressing the returned value in terms of the input parameters. Recall that in functional programming, one defines a function simply as being equals to a pure expression whose free variables correspond to the arguments.

Our language does support expressions with side-effects. However, instead of explicitly replacing each expression by a relation between states, we adopt a different approach. Since

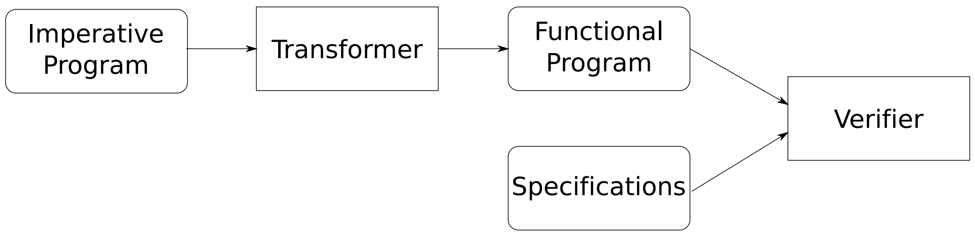

Figure 3.3: Transformation from imperative to functional before running the verifier.

functions do not have global side-effects, we encode each expression with local side-effects into an equivalent functional one. This way, we can then generate verification conditions in a straightforward fashion. Figure 3.3 illustrates our approach. Our goal is to take as input a program in the extended syntax and reduce it to an equivalent program in the core syntax. In the rest of this thesis, we will mix program expressions and logic formulas. As a convention, when we write the expression `e` we consider it as a program expression, while when we write e we consider its equivalent translation in first order logic.

The transformations are based on single static assignment and recursive functions to encode `while` loop. We detail the transformations in Chapter 4.

For the rest of the section, we assume that we applied our transformations to the input program. The resulting program consists only of the core language. We now give the exact construction of the verification conditions. Let us consider the following function:

```
@prec p
@post q
def f(x1, ..., xn) = e
```

where `p`, `q` and `e` are arbitrary expressions from the core language. The free variables of `p` and `e` are among `x1, ..., xn`. The postcondition `q` contains the same free variables as well as an additional `#res` variable which represent the returned value of `f`.

In the following we use the notation $\overrightarrow{x}$ to represent a vector of variables. We also use $p(\overrightarrow{x})$ as the formula where the free variables `x1, ..., xn` are substituted by $\overrightarrow{x}$ and we use $q(\overrightarrow{x}, r)$ in a similar fashion where `#res` is additionally substituted by r. Finally note that we will also freely mix `xn` with x_n depending on the context whether it is interpreted as a program expression or as a logic expression.

The formula expressing the correctness of the postcondition is as follows:

$$\forall \overrightarrow{x}.(p(\overrightarrow{x}) \implies q(\overrightarrow{x}, f(\overrightarrow{x}))) .$$

Dropping the universal quantifier and taking the negation, we get the formula F:

$$p(\overrightarrow{x}) \wedge \neg q(\overrightarrow{x}, f(\overrightarrow{x})) .$$

We have that F is satisfiable if and only if the function is not correct. This means that we can now ask an SMT solver for satisfiability of F, and if it returns that the formula is unsatisfiable then it means that the function meets its contract. Note that abstract functions do not need any treatment because they do not have an implementation.

A second set of verification conditions that we should verify for `f` is the one implying that each invocation of `f` respects the precondition `p`. So for each function in the program:

```
@prec p2
@post q2
def f2(x1, ..., xn) = e2
```

$$\frac{\texttt{e1}|c \longrightarrow s_1 \qquad \dots \qquad \texttt{en}|c \longrightarrow s_n \qquad \texttt{g} = \texttt{f}}{\texttt{g(e1, ..., en)}|c \longrightarrow (\bigcup s_i) \cup \{c \wedge \neg p(e_1, ..., e_n)\}}$$

$$\frac{\texttt{e1}|c \longrightarrow s_1 \qquad \dots \qquad \texttt{en}|c \longrightarrow s_n \qquad \texttt{g} \neq \texttt{f}}{\texttt{g(e1, ..., en)}|c \longrightarrow \bigcup s_i}$$

$$\frac{\texttt{b}|c \longrightarrow s_b \qquad \texttt{t}|c \wedge b \longrightarrow s_t \qquad \texttt{e}|c \wedge \neg b \longrightarrow s_e}{\texttt{if b then t else e}|c \longrightarrow s_b \cup s_t \cup s_e}$$

$$\frac{\texttt{b}|c \wedge i = e \longrightarrow s_b \qquad \texttt{e}|c \longrightarrow s_e}{\texttt{val i = e in b}|c \longrightarrow s_b \cup s_e}$$

$$\frac{}{\texttt{lit}|c \longrightarrow \emptyset}$$

Figure 3.4: Rules to compute the verification conditions of `f`.

Where `p2`, `q2`, and `e2` are defined respectively as `p`, `q`, and `e`.

Figure 3.4 shows the rules to compute the set of verification conditions for each expression. Applying these rules on each function `f2` while looking for invocations of `f` will collect all conditions on the preconditions `p`. More formally, we define the set S of verification conditions for `f2` by the following relation:

$$\texttt{e2}|p_2 \rightarrow S\ .$$

These rules describe how to collect the constraint on the argument of the function invocations. It starts from the function top level with the precondition being the initial constraint, then for each `if` expression, it visits both branches, each of them additionally constrained with the condition taken. Similarly, a `val` expression is essentially a let-binding and thus it constraints the identifier to be have the value of the right hand side. When the descent encounters a function invocation, if the function invoked is `f` then we add the current path constraint as one of the constraints on the arguments. Note that the actual verification condition should be:

$$\forall \overrightarrow{x}.(c \implies p)$$

where c is the current path constraint. But we already applied the transformation so that the verification condition can be sent to an SMT solver.

As a final note, the construction is exactly the same when `f` is abstract.

3.4 Solving the Verification Conditions

As seen in the previous section, we are left with a quantifier free term for which we need to decide satisfiability. Most of the involved theories are well understood and we can rely on the availability of efficient provers. The only difficulty is to solve expressions involving function invocations. In fact, non recursive function invocations can be easily solved by unfolding of the definition of the function. But recursive functions are more problematic, as it would require unbounded unfolding. This is one of the main distinction of our approach with respect to other techniques. Our verification conditions explicitly contain usage of function invocations, while in other techniques, one usually abstracts away the function invocations using their contracts.

We presented in Chapter 2 a technique that extends the reasoning capabilities of an existing solver with recursive functions. One difference with this algorithm is that we need reasoning for abstract functions. Such functions need special care when unrolling definitions in the procedure.

Unrolling of functions is done right after the main loop has tried both an over-approximation and under-approximation of the formula without successfully concluding anything. Adding clauses representing one unrolling of the function will refine the formula for the next steps. This involves introducing fresh variable for the body of the function as well as introducing a formula that makes the postcondition on the current parameters of the function holds. Abstract functions are handled in a similar way, however we do not add constraints for the body, since there is not any, but the postcondition is made to hold with fresh variables as well.

3.5 Properties of our Verification Procedure

In order to state formally the properties of our procedure, we must first formulate a few assumptions we made. We rely on the following three assumptions:

Termination. Each function defined in the program must terminate on all values that satisfy its precondition. The usage of a non-terminating function such as:

```
@post #res == 1
def f(x) = f(x) + 1
```

would introduce, after an unrolling step, a clause such as:

$$f(x) = f(x) + 1$$

which would make the formula unsatisfiable and hence implies that the function is correct, which cannot be the case since its postcondition is not valid.

Soundness of the underlying solver. The SMT solver used for deciding the over-approximated and under-approximated formulas should be sound and complete for the fragment we consider. Our method will only generate quantifier-free formulas in a theory that is known to be decidable. Hence, we assume that the underlying solver will always return a positive or negative answer for all queries, and that the answer it will return is correct.

Satisfiability of the `epsilon` constraint. We detail this limitation in Section 4.1, but essentially the predicate used as constraint for the `epsilon` construct should be satisfiable. If not, then the verifier will be able to prove validity of any property, similarly to the case with non-terminating functions. This assumption applies equally to the postconditions of abstract functions.

Under these assumptions, our procedure is complete for finding counter-examples. If one counter-example exists, it will eventually be found by our procedure. The reason being that the successive unrolling is done in a fair way, and that a counter-example must have a finite execution trace, which would be eventually detected by the unrolling procedure and the underlying solver would be able to satisfy it.

Our procedure is also sound, if it proves validity of the program, then it can be trusted. Similarly, if it finds a counter-example, the counter-example is a true counter-example. This last part can be actually dynamically tested by the solver, it can run the program with the counter-example found and check that the postcondition is indeed violated. However, such a test is not formally required.

Chapter 4

Program Transformations

Source-to-source transformations are semantic preserving mappings from the set of programs to itself. We make use of such mappings to eliminate some rich and complicated features in order to keep the core language minimal and easy to handle in our back-end. We first show how to represent `epsilon` primitives into abstract functions. We then discuss the encoding for imperative programs containing sequences of assignments and loops. Finally we discuss how we lift local functions to the top level.

Note that the order in which these transformations are applied is important. The order of presentation follows the actual order in which these passes are run. Each of these transformations actually depends on some properties having been eliminated by a previous pass or on a future pass eliminating a specific feature.

4.1 Non-Deterministic Choice Function

In this section we develop the meaning and utilization of `epsilon`. We first introduce the epsilon calculus, from which the `epsilon` primitive is inspired. We then justify the presence of `epsilon` in a programming language, including some use cases. Finally, we present how this primitive is handled in our verifier and conclude with some special notes on non-determinism.

4.1.1 Epsilon Calculus

David Hilbert has developed the epsilon calculus [47], which is syntactically similar to first order logic, except that the existensial and universal quantifiers are replaced by the epsilon operator, written ϵ. More precisely, if F is a formula and x a variable, then $\epsilon x.F$ is a term which is defined by the following axiom:

$$F(x) \implies F(\epsilon x.F)\ .$$

In other words, $\epsilon x.F$ return a value that can be assigned to x such that F becomes true. This is a non-deterministic operator in the sense that it can return any value that can make F true. If F is not realizable, then $\epsilon x.F$ can return any value.

Both existential and universal quantifiers can be encoded using ϵ:

$$\exists x.F(x) \equiv F(\epsilon x.F)$$

$$\forall x.F(x) \equiv F(\epsilon x.(\neg F))$$

4.1.2 Programming Primitive Based on Epsilon

This ϵ operator can be adapted to be part of a programming language and allow for constraint programming. Previous works [32, 33] have already introduced the possibility to use a `choose` function that have, basically, the same semantics as ϵ. Such a primitive can provide the programmer with a lot more expressive power.

At the same time, it allows to express an undefined choice such as a `random` function or input/output. For example, `random` can be defined as follows:

$$\texttt{def random()} = \epsilon x.\top$$

In our language, we define an `epsilon` primitive that takes a lambda expression as an argument. The lambda expression is to be interpreted as a predicate, and the `epsilon` expression returns any value that makes the predicate true.

The previous `random` example is thus written as follows:

```
def random() = epsilon(x => true)
```

4.1.3 Reasoning with Epsilon

Existing theories do not support ϵ as a part of the logic. Thus, we need to encode ϵ into some equivalent representation, before sending the formula to the solver. This is done via a program transformation, that we describe here.

So, given an expression `epsilon(x => e)` in some context $C : \texttt{expr} \longrightarrow \texttt{expr}$ with `e` being an arbitrary expression from the core language. We translate this to the following expression `e2`:

```
@post e
abstract def id() in id()
```

where `id` is a fresh identifier. We then plug `e2` back in the context, obtaining the final expression $C(\texttt{e2})$. We apply this transformation to every `epsilon` occurring in the program source. Note that this becomes a nested function in the place of the previous expression. Nested functions are not part of the core language, but in a later phase we will lift such functions to the top level. This notion of context is needed since `epsilon` is a pure expression that is found as a node in the abstract syntax tree of the language. As an example, in the following expression:

```
val r = epsilon(x => true) in
r + 1
```

the expression `epsilon(x => true)` is in the following context:

```
val r = _ in
r + 1
```

where _ is used as a placeholder for the missing expression. Applying the transformation would lead to:

```
@post true
abstract def e1() in e1()
```

and plugging the expression back in the context finally gives:

```
val r =
  @post true
  abstract def e1() in e1()
in r + 1
```

The transformation is valid since the semantics of the abstract function is chosen to match the one expected by `epsilon`.

Soundness with Satisfiable Predicate

Since the `epsilon` predicate is translated to a postcondition of an abstract function, the handling of abstract functions in the back-end decision procedure needs some discussion. As described in Chapter 3, an abstract function is replaced by a fresh variable that is additionally constrained by the postcondition.

In particular, if the postcondition is invalid, this will makes the overall unrolling procedure returns unsatisfiability of the formula. This translates into any property being concluded as valid. Let us consider an example:

```
@post false
def f() = epsilon(x => x != x)
```

Since the postcondition is `false`, the example should be invalid.

One unrolling leads to the following formula:

$$\top \wedge v_x \wedge v_x \neg v_x$$

where v_x is the fresh variable introduced for `x`. This formula is obviously unsatisfiable because of the last clause. This means that no counter-example is found and that the actual property is valid, which is not sound.

Our procedure then has the stated soundness and completeness only when `epsilon` is used with satisfiable predicates. One solution would be to introduce existential quantifier and solve a general condition that state the existence of a solution to the predicate. However, the drawback to this solution is that we will lose all some other properties since the formula would no longer be quantifier-free and the underlying solver will not be complete anymore.

4.1.4 Non-Determinism

Functions from the input language are eventually translated into mathematical functions in a logical theory. Mathematical functions are formal and precise objects, and in particular they are deterministic, that is, given the same input, they will return the same output.

Currently our transformation directly maps `epsilon` to a corresponding fresh deterministic function. In that sense, our transformation enables constraint programming, where one only specify the result and not how to compute it, but it does not provide direct non-determinism.

To see the problem let us consider the following program:

```
def rand() = epsilon(x => true)

@post true
def wrong() = rand() == rand()
```

Since `rand` is supposed to be non-deterministic, we would expect that the postcondition of `wrong` is invalid. However, with the current construction we defined, this program is actually valid. To understand this result, we should examine the encoding that is finally send to a prover. Three functions are defined: `rand`, `e1` and `wrong`, each one taking no argument.

Then the following formula represent the unrolling of the functions:

$$\texttt{rand()} = \texttt{e2()} \wedge (\texttt{wrong()} \iff \texttt{rand()} == \texttt{rand()}) \wedge \texttt{true}$$

Where the final `true` is the constraint on the value of `e2()` and can be safely ignored, which means `e2()` can take any value. However, as all functions in math are deterministic, this results in `e2()`, as well as `rand()`, always taking the same value.

One way around the problem, is to explicitly add additional parameters to the functions. Let us consider this modified version of the previous program.

```
def rand(a) = epsilon(x => true)

@post true
def wrong() = rand(0) == rand(1)
```

The generated functions are still the same, but `rand` and `e1` will this time take one argument:

$$\texttt{rand(a)} = \texttt{e2(a)} \wedge (\texttt{wrong()} \iff \texttt{rand(0)} == \texttt{rand(1)}) \wedge \texttt{true}$$

Since we still have no constraint on the value taken by `e2(a)`, the solver can choose any value, in particular it can choose a different value for each `a`. Thus, in that example, it will be able to find a model that contradicts the postcondition of `wrong`, for example by assigning `e2(0) = 0` and `e2(1) = 1`.

Note that, even though it looks like you need to use statically known constants in order to generate non deterministism, this can be made to work in more complicated situation, like unbounded recursion:

```
def rand(a) = epsilon(x => true)
def rec(i, a) = if(i == 0) 0 else rand(a) + rec(i-1, a+1)
```

The trick is simply to build an implicit sequence of different values for the argument to the function we wish to make non-deterministic. It is up to the programmer to encode the correct behaviour that he wishes to have.

Alternative Semantics

One might wonder why we chose this deterministic semantics instead of automatically encoding the non-determinism. For one, nothing is taken from the user, he can still encode non-determinism if that is what he truly wishes. In fact, our approach actually allows the programmer to have access to a deterministic function if he wishes so. In some situation this could actually be what he wants.

We considered two ways to try to automate the encoding of `epsilon` in a non-deterministic equivalent:

Specific handling of `epsilon` in the decision procedure. This technique involves automatically generate fresh variables to replace `epsilon` expressions while introducing new unrolling of recursive functions. There is also a need to track which `epsilon` has been replaced by which variable in order to build a model at the end. It is particularly cumbersome to do so when `epsilon` appears in a recursive function and at each level corresponds a different choice of value. It would seems that we need to record some sort of stack trace along with the program point of the `epsilon` being instantiate.

This would also be dangerous to keep `epsilon` as a non-deterministic expression that far into the system. For example let expansion are very dangerous because we would need to somehow record that expanded `epsilon`s expressions were the same expression statically and need to return the same value. This is different from all the other features of the core language, that are purely functional and hence can be manipulated more freely.

Adding the extra variables during a transformation pass. In this case, we add another source-to-source transformation that would automatically do the manual encoding we presented above. This has the advantage of being a very modular approach, one could enable or disable the pass as needed. It is also entirely orthogonal to the rest of the system. We do not see any immediate drawback to this approach, but we did not have enough time to implement it. It is definitely worth exploring in the future.

4.2 Imperative to Functional

We handle local variables, assignments, loops, and sequencing by transforming them to equivalent functional code. By local variables, we mean using standard mutable variables only in bodies of functions. The functions can be written with an imperative style, but with no global side effect. Thus, all functions are pure, and there must be an equivalent purely functional definition of the function.

Recall that variables declarations, assignments, loops, and sequencing are extensions to our core language. Our aim is then to map any program using these extensions into a program that makes use of the functional core only.

4.2.1 Imperative Code as Transformations of States

It is well known that any imperative program can be simulated by explicitly carrying a program state and returning the new state along with every statements. This is a very general approach, but it does not always generate intuitive code and it is not the most efficient one. Another work has already shown how to write deductive rules to do such transformations and mechanically proving their correctness [48]. The work done here has a similar flavor but has been done independently and focuses on practicality rather than formalism.

In our language, the state of the program is simply an assignment of program variables to values, and one way to update this state information is to introduce new names as let bindings each time a variable is updated and to keep a mapping from program variables to their current names. This is essentially transforming the program into an SSA form. It has already been established that SSA form is equivalent to functional programming [5].

We present a recursive procedure to map imperative statements to a series of definitions (`val` and `def`) that form a new scope introducing fresh names for the program variables, and keeping a mapping from program variables to their current name inside the scope. The procedure is inspired from the generation of verification conditions from a program [18, 23, 49]. However such methods suffer from an exponential growth in the size of the program fragment. In some sense, our transformation to functional programs, followed by a later generation of verification conditions avoid the exponential growth similarly to the work of Flanagan et al. [21].

Intuitively, we can represent any imperative snippet as a series of definitions followed by a group of parallel assignments. These assignments will rename the program variables to their new name, that is, the right hand side will be the new identifiers of the program variable (that have been introduced by the definitions) and the left hand side will be the program variables themselves. These parallel assignments are an explicit representation of the mapping from program variables to their fresh names. As an example, consider the following imperative program:

```
x = 2
y = 3
x = y + 1
```

It can be equivalently written as follows:

```
val x1 = 2 in
  val y1 = 3 in
    val x2 = y1 + 1 in
      x = x2
      y = y1
```

This is the intuition behind this mapping from program variables to its fresh representations. The advantage is that we can build a recursive procedure and easily combine the results when we have sequences of statements.

4.2.2 Examples

To give the reader some intuition on how such transformation could work, we give some simple examples. Let us consider the following program:

```
def foo(x) =
  var y = 3 in
    var z = x in
      if(y < z)
        y = y + 1
      else
        z = z + 1
      y + z
```

The function `foo` is pure but its implementation is using imperative features. However it can equivalently be rewritten as follows:

```
def foo(x) =
  val y1 = 3 in
    val z1 = x in
      val t1 =
        if(y1 < z1)
          val y2 = y1 + 1 in (y2, z1)
        else
          val z2 = z1 + 1 in (y1, z2)
      in t1._1 + t1._2
```

We can see that the `if` construct is automatically transformed to its functional version returning the tuples of modified variables.

As another example, containing loops, let us consider the following program:

```
def foo(x) =
  var y = 0 in
    var i = 1 in
      var n = 100 in
        while(i <= n)
          y = y + x
          i = i + 1
        y
```

Again we have the following equivalent definition of the function:

$$\frac{}{\texttt{x = e}\longrightarrow\texttt{val x' = e in \_ }|\{x\to x'\}}$$

$$\frac{\texttt{b}\longrightarrow s_b|m_b \qquad m=\{x\to x'\}}{\texttt{var x = e in b}\longrightarrow(\texttt{val x' = e in \_ })\circ m(s_b)|m\cup m_b}$$

$$\frac{\texttt{e1}\longrightarrow s_1|m_1 \qquad \texttt{e2}\longrightarrow s_2|m_2}{\texttt{e1 e2}\longrightarrow s_1\circ m_1(s_2)|m_1\cup m_2}$$

$$\frac{\texttt{t}\longrightarrow s_1|m_1 \quad \texttt{e}\longrightarrow s_2|m_2 \quad \mathrm{dom}(m_1\cup m_2)=\{\overrightarrow{x}\} \quad \texttt{t'}=s_1(m_1(\overrightarrow{x})) \quad \texttt{e'}=s_2(m_2(\overrightarrow{x}))}{\texttt{if(c) t else e}\longrightarrow\texttt{val }\overrightarrow{x_2}=\texttt{if(c) t' else e' in \_ }|\{\overrightarrow{x}\to\overrightarrow{x_2}\}}$$

$$\frac{\texttt{e}\longrightarrow s|m_1 \qquad m_1=\{\overrightarrow{x}\to\overrightarrow{x_1}\} \qquad m_2=\{\overrightarrow{x}\to\overrightarrow{x_2}\} \qquad \texttt{s'}=m_2(s)}{\texttt{while(c) e}\longrightarrow\texttt{def w}(\overrightarrow{x_2})=\ \texttt{if}(m_2(c)) \quad s'(w(\overrightarrow{x_x}))\ \texttt{else }\overrightarrow{x_2}\texttt{ in val }\overrightarrow{x_3}=\texttt{w}(\overrightarrow{x})\texttt{ in \_ }|\{\overrightarrow{x}\to\overrightarrow{x_3}\}}$$

Figure 4.1: Transformation rules for imperative programs.

```
def foo(x) =
  val y1 = 0 in
    val i1 = 1 in
      val n1 = 100 in
        def w1(y2, i2) =
          if(i2 <= n1)
            val y3 = y2 + x in
              val i3 = i2 + 1 in w1(y3, i3)
          else
            (y2, i2)
        in val t1 = w1(y1, i1) in t1._1
```

In that case, we have eliminated the `while` loop by mapping it to a recursive function. The loop body consists of a check for whether the loop condition is met, and, if so, we compute the new value and recursively call the function. Whenever the conditions becomes false, then we return from the function with the current computed values.

4.2.3 Formal Transformations Rules

Figure 4.1 shows the formal rules to rewrite imperative code into equivalent functional code. The rules define a relation $e \longrightarrow s|m$ between an expression e, a function from expressions to expressions s (a scope), and a substitution map m for variable names. We give a rule for each different imperative construct. This is a mathematical formalization of the intuition of the previous sections, we defined a scope of definitions as well as maintained a mapping from program variables to fresh names. Note that each time we introduce some primed or subscribed version of the variable, we are implicitly adding a newly fresh variable.

If e is an expression, s a scope and m a substitution map, we will write $s(e)$ the resulting expression after introducing e into the scope s (this can be thought as a function application). We will denote $m(s)$ the scope s' where, for all variable x such that $x \to x' \in m$ then x is replaced by x' in s'. Similarly, we use the notation $m(e)$ for the expression with the substitution applied to the expression e. If m' denotes another substitution map, then we denotes the update of m by m' with $m \cup m'$. That is, in case the same variable is mapped in both m and m', the mapping of m' will override the one of m. We will represent the scopes using an expression in the formal syntax of the language and with an _ in place of the missing expression. We use the $\circ$ operator to compose two functions as in the usual mathematical way.

For ease of presentation, we will assume that blocks of statements are terminated with a pure expression r from the core language, which is the returned value. So, given the initial body of the block b and the following relation:

$$b \longrightarrow s|m$$

we can define the function expression equivalent to `b; r` by:

$$s(m(r)) \ .$$

This simplification allows us to ignore the fact that each of those expression with side effect actually returns a value, and could be the last one of a function. This is particularly true for the `if` expression which can return an expression additionally to its effects. The rules can be generalized to handle such situation by using a fourth element in the relation denoting the actual returned value if the expression was returned from a function or assigned to some variable. Note that in our system we have implemented this more general behaviour. We have also assumed that expression such as right hand side of assignment and test conditions are pure expression that do not need to be transformed. However, it is also possible to generalize the rules to handle such expressions when they are not pure, but the presentation would become very hard to follow so we will not consider it. Again, in our implementation we support this more general transformations.

The inductive property we need to reason about these rules are as follows. We maintain the property that the scope introduces a series of fresh variables names that defined the new value of the variables in terms of the original variables. The mapping is used to carry the information of which fresh variable contains the current value of which original variable. This should explain how the scopes from subexpressions must be applied to each other when combining expressions. Let us examine these rules in detail.

The first two rules on variable declaration and assignment are both similar and relatively straightforward. The new scope consists of a new let-binding with a fresh name. This fresh variable is also used as the mapping.

The sequencing operator `e1 e2` is already more interesting. Note that we only consider a bock of 2 expressions, but a block of n expressions can always be encoded into sub-blocks of 2 expressions. In particular, it will help to illustrate how we can combine scopes from subexpressions. Consider the following example:

```
x = 3;
x = x + 1
```

And the following subderivations:

$$\frac{}{\texttt{x = 3} \longrightarrow \texttt{val x1 = 3 in \_}|\{x \rightarrow x_1\}}$$

$$\frac{}{\texttt{x = x + 1} \longrightarrow \texttt{val x2 = x + 1 in \_}|\{x \rightarrow x_2\}}$$

First we can compute $m_1(s_1)$ = `val x2 = x1 + 1` and $m_1 \cup m_2 = \{x \rightarrow x2\}$. Thus we obtain the following relation:

$$\texttt{x = 3; x = x + 1} \longrightarrow \texttt{val x1 = 3 in val x2 = x1 + 1 in \_}|\{x \rightarrow x_2\}$$

which is as expected. It introduces two intermediate let-bindings in order to compute the new value of `x` at the end, which is represented by `x2` as the mapping indicates.

Now let us look at the two most interesting rules, the ones for `if` and `while`. Let us consider the following example:

```
if(x < 0)
  x = x + 1
else
  y = y + 1
```

We derive the relation for both cases:

$$\texttt{x = x + 1} \longrightarrow \texttt{val x1 = x + 1 in \_}|\{x \to x_1\}$$

$$\texttt{y = y + 1} \longrightarrow \texttt{val y1 = y + 1 in \_}|\{y \to y_1\}$$

Now the expression $\mathrm{dom}(m_1 \cup m_2)$ denotes the domain of the union of both substitution map. This corresponds to the set of original program variable that are assigned a new value in either of the two branches. In our example, this would be computed as follows $\mathrm{dom}(\{x \to x_1\} \cup \{y \to y1\}) = \{x, y\}$.

In the rules we use $\overrightarrow{x}$ in a very flexible way, in particular in the domain expression above it is used to captured all variables that are modified. It is then use as a tuple expression in $s_1(m_1(\overrightarrow{x}))$. So in our example we can compute both `t'` and `e'`, using $\overrightarrow{x} = (x, y)$, $m_1((x, y)) = (x1, y)$, and $m_2((x, y)) = (x, y1)$, as follows:

$$\texttt{t'} = s_1(m_1(\overrightarrow{x})) = \texttt{val x1 = x + 1 in (x1, y)}$$

$$\texttt{e'} = s_2(m_2(\overrightarrow{x})) = \texttt{val y1 = y + 1 in (x, y1)}$$

As we can see, both `t'` and `e'` are now pure expressions and can be use as `then` and `else` branches of the functional `if`. Combining these derivations, we get the final scope:

```
val (x2, y2) = (
  if(c)
    val x1 = x + 1 in (x1, y)
  else in _ }
    val y1 = y + 1 in (x, y1)
) in _
```

which introduces fresh variables for the program variables. The final substitution maps the program variables to these fresh variables.

Finally the rule for `while` introduces a recursive function to replace the loop. The transformation is relatively intuitive but care must be taken with the various substitutions to apply them correctly. One recursive call to the function emulates one iteration of the loop. Once again an example will better describe the rule:

```
while(x < n)
  s = s + x
  x = x + 1
```

Once again we can derive the relation for the two assignments:

$$\texttt{s = s + x} \longrightarrow \texttt{val s1 = s + x in \_}|\{x \to x_1\}$$

$$\texttt{x = x + 1} \longrightarrow \texttt{val x1 = x + 1 in \_}|\{x \to x_1\}$$

And combine them:

$$\texttt{val s1 = s + x in val x1 = x + 1 in \_}|\{x \to x_1, s \to s_1\}$$

Which gives us the value for s and m. Now we can compute s':

```
val s1 = s2 + x2 in val x1 = x2 + 1 in _
```

Finally we define the recursive function as:

```
def w(x2, s2) =
  if(x2 < n)
    val s1 = s2 + x2 in val x1 = x2 + 1 in w(x1, s1)
  else
    (x2, s2)
```

And the rest of the body consists of the initial function call to `w` with the original variables and assigning the result to some additional fresh variables.

4.3 Lifting of Nested Functions

The transformation for `while` loops from the previous section introduced some nested definitions of functions. The possibility to define functions in any local scope is a very powerful and useful feature of functional programming. However, the core language only supports top-level definitions of functions, thus the need to lift all such functions.

Techniques from compilers constructions to solve this problem are well known [4]. These techniques have to deal with the difficult problem of accessing local mutable variables after having extracted the function from its initial scope. On the other hand, since our previous transformation has eliminated all imperative constructs, we are left with pure functions that only accessed immutable variables from the local scope.

The process can be make clearer if we further separate it into two distinct phases. In the first one, each function is closed with the relevant variables in scope. Then, in the second phase, we can hoist each function to the top level.

4.3.1 Function Closure

Nested functions can read variables that are defined in an enclosing scope, for example the formal parameters of its hosting function or a let-binding in an enclosing function. The function closure step consists in augmenting the signature of functions with all variables that are used by the function but not defined in its formal parameters or body. It also consists in updating all corresponding function invocations accordingly.

Since we do not handle global variables, all top level functions only access variables that are defined locally, either in its formal parameters or in some let-bindings in its own body, so this step only applies to nested functions.

We need to consider three different expressions that can access variables in the current scope for each function:

- The precondition
- The body
- The postcondition

The precondition needs some special care. In particular, any precondition of enclosing functions should still holds in the nested function.

First we must be able to compute all free variables accessed by an expression. Figure 4.2 shows how to compute the set of free variables in an expression. The rules are not very complicated and should be easy to understand. Given an expression e, if $e \longrightarrow V$ holds, then V is the set of free variables in e. The rules do not mention how to handle pre/postconditions of nested functions definitions. Their extensions simply requires the same recursive application to the contract, while removing the formal parameters and, in the case of the postcondition, the result variable.

As mentioned above, the precondition of the function needs special care. Indeed, the function being defined locally and making use of some formal parameters of enclosing

$$\frac{\texttt{e} \longrightarrow s_e \qquad \texttt{b} \longrightarrow s_b}{\texttt{val x = e in b} \longrightarrow s_e \cup (s_b \backslash \{\texttt{x}\})}$$

$$\frac{\texttt{e1} \longrightarrow s_1 \qquad \texttt{e2} \longrightarrow s_2}{\texttt{def f(x1, ..., xn) = e1 in e2} \longrightarrow (s_1 \backslash \{\overrightarrow{x}\}) \cup s_2}$$

$$\frac{\texttt{e1} \longrightarrow s_1 \qquad \dots \qquad \texttt{en} \longrightarrow s_n}{\texttt{f(e1, ..., en)} \longrightarrow \bigcup s_i}$$

$$\frac{\texttt{t} \longrightarrow s_t \qquad \texttt{e} \longrightarrow s_e}{\texttt{if(c) t else e} \longrightarrow s_t \cup s_e}$$

$$\frac{}{\texttt{x} \longrightarrow \{\texttt{x}\}}$$

Figure 4.2: Rules to compute the set of free variables.

$$\frac{\texttt{e}|c \longrightarrow m_e \qquad \texttt{b}|c \wedge x = e \longrightarrow m_b}{\texttt{val x = e in b}|c \longrightarrow m_e \cup m_b}$$

$$\frac{\texttt{e1}|c \wedge p \longrightarrow m_1 \qquad \texttt{e2}|c \longrightarrow m_2}{\texttt{@pre p def f(x1, ..., xn) = e1 in e2}|c \longrightarrow \{f \rightarrow c \wedge p\} \cup m_1 \cup m_2}$$

$$\frac{\texttt{e1}|c \longrightarrow m_1 \qquad \dots \qquad \texttt{en}|c \longrightarrow m_n}{\texttt{f}(e1, ..., en)|c \longrightarrow \bigcup m_i}$$

$$\frac{\texttt{t}|b \wedge c \longrightarrow m_t \qquad \texttt{e}|\neg b \wedge c \longrightarrow m_e}{\texttt{if(b) t else e}|c \longrightarrow m_t \cup m_e}$$

$$\frac{}{\texttt{x}|c \longrightarrow \{\}}$$

Figure 4.3: Rules to collect path constraints.

functions, the precondition from such enclosing functions on the formal parameters are implicitly true in the nested function. After the closing phase, we introduce new variables to replace the ones coming from an enclosing scope, which, as a results, loses all connections to the original variables and their constraints.

We give rules to associate each nested function with its extended precondition in Figure 4.3. Each constraint is constructed from the enclosing functions and the conditions taken along the computation path.

So let us consider the following nested function expression:

```
@pre p
@post q
def f(x1, ..., xn) = b in e
```

Using rules from Figure 4.3, we can derive the precise precondition `p'` that applies to `f`. Now, using the rules from Figure 4.2, we can compute $V = V_1 \cup V_2 \cup V_3$ such that `p'` $\longrightarrow V_1$, `q` $\longrightarrow V_2$ and that `b` $\longrightarrow V_3$. Note that we need to compute the free variables of `p'` and not simply `p`.

Finally we introduce fresh variables for each variable in V and build the map $m = \{v \rightarrow v' | v \in V \wedge v' \text{ is fresh}\}$. We can now build the final nested functions:

```
@pre m(p')
@post m(q)
def f(x1, ..., xn, v1', ..., vm') = m(b) in e
```

where `v1', ..., vm'` are the newly introduced fresh variables.

This function `f` has a new signature. This means that all invocations of this function are now wrong. There are two places where invocations of `f` can occur: inside the new body of `f` or inside the expression `e`. The function invocations inside the body need to be extended with the variable `v1', ..., vm'`, while the function invocations inside the expression `e` needs to be extended by the original versions of these same variables: `v1, ..., vm`.

4.3.2 Function Hoisting

Once the formal parameters of the function have been augmented with some fresh identifier for each variable in scope, the function can be moved to the top level without modifying the meaning of the program. Indeed, as long as the function remains visible in the scope where it is used, its definition is now complete and no longer requires access to some external variables.

Since moving the function to the top level will only increase visibility of the function, the only risk would be that this function is now used by part of the code that should previously not be able to access it. This does not happen since we use unique identifiers for every single definition, thus preventing any form of overloading of existing functions, and none of our transformation will introduce code that would call this function from an illegal place.

Losing Call Sites Information

A subtle loss of precision happens when we hoist functions that were previously only defined locally. When a function is defined locally in a scope, it is available for a finite number of program point. On the other hand, when a function is defined at the top level, it is implicitly assumed that it will be available for the whole program, and could even be used by an external component. This also translates in a different way of reasoning about these two classes of functions. When a function is defined locally, we can collect all of its call sites, because there are a finite number of them and there will not be any new ones. We say that the scope in which the function is defined is closed, by opposition to the open nature of the top level. In essence, we would be able to assume that the parameters of the functions can only take values consistent with all the call site. For example, in the following:

```
def f(x) =
  @post #res == 3
  def g(y) = y in
  g(3)
```

the local function `g` is only called once with the value `3` and hence it is possible to conclude that it will always return the value `3`, hence proving the postcondition.

Of course, when we hoist such function at the top level, the information is lost. Even when the complete program is given, and all the call sites are known, it is not natural to constraint the parameters of all functions in a consistent manner with its call sites. This is definitely not the intended behaviour. In fact, we often define top level functions that are never called just to express some general properties.

Knowing all of the static call sites is not sufficient, some of these could be recursive calls to the local function. Hence we would have to solve a fix-point to find the correct precondition on the parameters. This loss of precision happens mostly with the transformation of loops:

```
def f(x) =
  var i = 0 in
  while(i < 10)
    i = i + 1
  i
```

such program is translated to:

```
def f(x) =
  val i = 0 in
  def w(i1) = if(i1 < 10) w(i1+1) else i1 in
  w(i)
```

where there are two different call sites of the `w` function. It is easy to see that `w` is always called by an integer ≥ 0, but automatically detecting this fact would require computing a fix-point. However, from the original loop, this is clearly a valid invariant, and it is lost when hoisting `w` at the top level.

Chapter 5

Test Cases Generation

As a complementary method to automatically proving properties, we propose a technique to generate test cases to give some good evidence of correctness. Manual testing can be very tedious and cannot exercise all possible program executions. This is a fundamental limitation of program testing: there is, in general, an infinite number of different executions. When one wishes to prove correctness of a program, one must ensure that this infinity of executions traces does not contradict the specifications, which is of course not doable by enumeration. However we can relax this requirement and only target a coverage of statements, which are known statically and are of a finite number.

The general methodology is as follows:

1. Write the program with its specifications.
2. Try to apply the verification algorithm. If the program is not correct, then the procedure, being complete for counter-examples, returns a faulty input.
3. If the program is correct, then either the verification procedure will be able to prove so and return that the program is correct, or it will loop forever.
4. In case it loops for some time, then timeout the procedure and apply the test cases generation algorithm.
5. This provides a collection of test cases that can be executed and checked against the specifications, if none of them exhibits a faulty behaviour, then it provides strong evidence for the correctness of the program.

We apply the test cases generation phase after having mapped programs into their functional representations. That means we can assume a simple language for the generation of test cases while still being able to use it on complex imperative programs.

5.1 Graph Representation of Functional Programs

We want to capture the inter-procedural behaviour of the program. For this, we generate a graph representation of programs that is very close to a control flow graph. In particular, we do not abstract away the functions. Instead, we have a global graph in which each function is completely inlined.

We are interested in representing the control flow across the program. If this control flow is well defined and understood in the case of imperative programming, it is much less obvious in the case of functional programming. This comes mostly from the fact that, in functional programming, there is no notion of sequence of operations, and the computation is defined by the evaluation of a single, composite, expression. Basically, there is no such thing as a statement in functional programming, while a usual imperative program consists in a sequence of statements, where the flow of execution goes from statement to statement.

We quickly discuss the evaluation order of our core language. The semantics is call-by-value and arguments to function invocations are evaluated from left to right. Conditional expressions only evaluate the branch that is taken. This is arguably the most natural evaluation strategy and is definitely the most standard one. It also matches the one of Scala.

In terms of control flow, except in the case of `if` expressions, when an expressions is evaluated automatically all of its subexpressions are evaluated. Aiming at a complete statement coverage translates to find several input values so that all of the expressions are evaluated at least once. As a minimal example, if we have the expression:

```
if(x < 0) -x else x
```

then there are four expressions total: `if(x < 0) -x else x`, `x < 0`, `-x`, and `x`. On input `x = -1`, three of these expressions are evaluated, missing the `else` branch `x`. Complete coverage can be obtained using a second input of `x = 1`.

We build a directed graph $G = (V, E)$, with $E \subseteq V \times V$, from a program P written in the core language. The set of nodes V will correspond to program points, while the set of edges E will represent possible transition from a program point to another one. The program P is simplified by expanding all let-bindings. For each function defined in P, we add a node in V representing the entry point to the function, labeled with its signature. Then we decompose the expression that defined the function into its subexpressions and add one node for each subexpression, as well as one node for the complete expression. We introduce a directed edge from the parent to each its subexpressions. Each node is labeled with the expression it represents. Note that contrary to control flow graphs for imperative programs, edges do not need any label such as assignment because the entire computation is contained in the expression. We can think of such a graph as a tree representation of the program using algebraic data types. It should be noted that this graph contains enough information to evaluate the program, such that we can ignore the original source code.

One possible interpretation of the graph V is that, starting from an entry point, the execution of the program goes through each node that are reachable following the edges. This is true as long as the edge does not correspond to a branch of an `if` expression. To make this more explicit, we will label each edge with the conditions that need to be true in order to be able to evaluate the expression at the other end of the edge. These conditions are simply the conditions of the `if` expressions and their negations. All of the other edges can be labeled with `true` since they will always be taken.

We will assume that one function is marked as the main function. The node corresponding to this function will then be considered as the starting node in the graph. Now, our goal is to find input values to this main function so that every single nodes in the graph is visited. The difficulty lies in how to handle joint paths. It is not sufficient to consider each function independently, since it could be the case that one path in the callee is infeasible because of a condition taken in the caller. For example, let us consider this program:

```
def f(x) = if(x < 0) g(-x) else g(x)
def g(y) = if(y < 0) -y else y
```

where `f` is the main function. Figure 5.1 shows the graph for this program. Here, there are two paths through `f`, each of them will then be joint at the entry point of `g`. The conditions along these paths will force the argument of `g` to be greater than 0, which in turns will force to take the `else` in `g` and makes the path through the `then` branch infeasible. If we

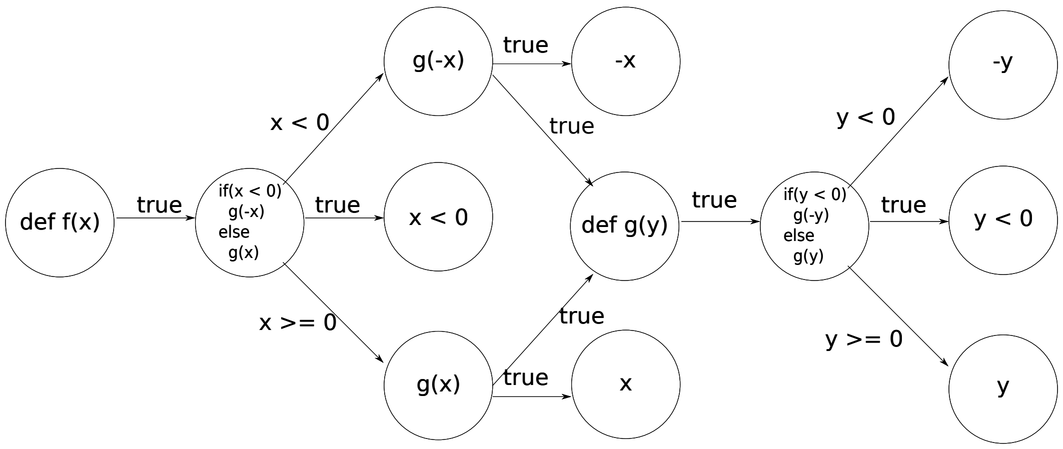

Figure 5.1: Complete control flow graph of a functional program.

would have considered `g` on its own, then we could have found two possible inputs (for example, `y = -1` and `y = 1`) that could cover both branches.

We are interested precisely in constructing all possible paths starting from a specified entry point. This is a realistic settings as programs usually always start from a single entry point. This is actually an important restriction, because some errors that could be obtained using inputs to some internal functions might actually not be reachable when starting from the entry point.

We would like to be able to regard a path in this graph as exactly an execution trace. This is unfortunately not true in the current graph, because many nodes could have multiple subexpressions, all of them being automatically evaluated. So, if we look at one particular path, we will visit only a certain number of nodes and ignore some other nodes that would actually be visited if the execution would have taken this graph.

A final refinement to the graph representation will allow us to compress it and prune many sections of the program that are automatically covered. An expression such as:

```
f(a + b, c - d, e)
```

contains many subexpressions, but all of them are automatically evaluated each time because they are not under a branch of an `if` expression. These nodes and edges are easy to spot: an edge that is labeled with `true` is an edge that is always taken and when two nodes are connected with such an edge we can simply merge them together. We can see in Figure 5.2 how the previous example is simplified.

At that point, the graph is fairly simple, an edge will either be labeled by a condition and would correspond to an `if` expression, or it will be an edge connecting a function call to the corresponding entry point. However, we still do not have a one-to-one correspondence between paths in the graph and execution traces. For example, an expression such as:

```
f(g(x,y), z)
```

has no condition and thus would be compressed to a single node with two outgoing edges, one towards `f` and one towards `g`. When taking a path in this graph, we have a choice among the two edges, however, in term of real execution, both are automatically taken.

5.2 Enumerating all Paths

As discussed in the previous section, this graph representation does not quite provide us with a good enough abstraction to view graph paths as execution paths. But, despite

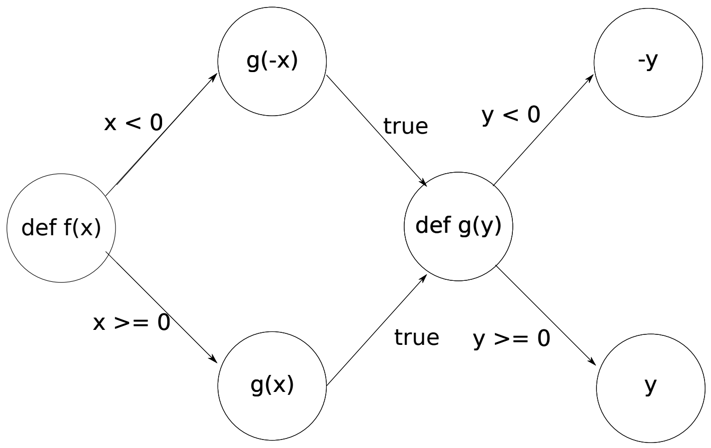

Figure 5.2: Simplified control flow graph of a functional program.

```
f(e1, ..., if(b) t else e, ..., en) ⟶
if(b) f(e1, ..., t, ..., en) else f(e1, ..., e, ..., en)
```

Figure 5.3: Rule to hoist `if` expressions.

not being a perfect correspondence, if we are able to follow one path in the graph and satisfy the constraints along the path, we have really covered all nodes out of this path. In fact, we can then start the visit again from the start with the values that satisfy the constraints, and mark all possible nodes that are reachable given these values. Applying such idea successively to many paths should give us a complete coverage of all node in the graph, and hence a complete coverage of all statements/expressions in the original program.

There are two different features of the graph that we need to consider. When a node has an edge labeled with a condition, we need to record it and try to visit both eventually. When we encounter a function invocation node, we will need to follow the edge to the entry point of the callee and record the new value of the formal parameters. However, we must be careful at that point as the arguments to the function call can still be complex expressions and may actually contain other function invocations or even `if` expressions that would require more work. We cannot simply continue the search in the new function and forget about the rest of the computation in the current function.

To handle this problem at the graph level seems overly complicated, it would require keeping track of many branching choices made while backtracking at the function invocations. Rather, we prefer to pre-process the program in order to make the resulting graph much easier to reason about. The rewriting rules we apply to the program is shown in Figure 5.3. Its role is to find `if` expressions that are subexpressions of some other expressions and to move them at a higher level. Applying these rules until convergence will guarantee that the final program will have all its conditional expressions at the top level, and only function invocations or other variables or literals as leafs.

Given this simplified graph, the problem of statements coverage can be reduce to a search in a graph. One node roughly corresponds to one expression in the original program. All the nodes that have been removed from the graph are automatically visited given that some particular node in the compressed graph is visited. Since at the end we aim to visit all of these nodes, the removed nodes are no longer relevant. Even if we were not able to visit all the nodes, a node that has been removed had only one entry point, and that entry point will still be present. So, if it is not reachable, then the node that has been removed was also not reachable. We do not lose any precision by simplifying the graph.

Since we want to cover all program statements starting from a main function, in the graph we will want to visit all nodes starting from an initial node, the node corresponding to the entry point of the main function. Of course, it might not be possible to visit all paths in one path. And in fact, since we are not interested in visiting more than once the same node, we will only try to build simple paths. Simple paths are paths without cycle, in term of execution that means no recursion. Assuming at least one `if` expression, we will need to find several paths in order to cover all of the nodes.

A path is a sequence of nodes and edges, each edge being labeled by a condition formula. Note also that some edges will introduce a mapping from variable to some new values (the arguments to a function invocation). When we have a complete path, we want to build a formula out of all the conditions and then ask our decision procedure for a solution. If a solution is found, the path is feasible and we obtain input values to the main function that will force the execution along this path. In fact, it will even visit more than just these nodes, but at the very least we will have coverage of this path.

Now the last point that needs discussion is the enumeration of these paths. We apply a depth first search, stopping the search when either we reach a dead-end or we visit a node that was previously visited by the *current* path. It is not correct to stop as soon as we visit a node that was already marked by another path. Each time we terminate a search, we record the path, backtrack and continue the search along a different direction. There is always a final number of simple path in a finite graph, hence this enumeration eventually terminates.

Each path can then be combined in a formula and we can ask for satisfiability. Note that we still need reasoning about recursive functions in the back-end solver. This is because condition are often expressed in term of function defined in the program. If the formula is satisfiable, then we found an input that will visit exactly this path and we can mark all nodes as visited. On the other hand, if the formula is unsatisfiable, then we will have to find another path in order to visit these nodes.

Our enumeration lists all simple paths that terminate at a leaf or that loop. However, there are many more simple paths. In particular, all initial segments of the simple paths we just listed are other possible candidates to an execution trace. It is safe to ignore those, this follows because if such an initial segment is satisfiable, it must be the case that there exists one extension to this path that is also feasible. It comes from the fact that whenever an edge is labeled with a condition, there is a second edge labeled with its negation and hence at least one of those can be taken. This is by construction of the program, because `if` expressions introduce disjoint conditions that form a partition.

Now, our enumeration ignores cycles, so we need to make sure it will not miss some nodes that could only be reached by first cycling in the graph. The key observation here is that cycling in the graph means recursive function call, and, assuming termination, it must be the case that the sequence of function calls is monotonously decreasing. Basically, it must be the case that given any value we chose, the terminal branch is eventually taken. It is possible to construct such examples:

```
def f(x) = if(x <= 0) g(-x + 1) else g(x)
def g(x) = if(x <= 0) 1 else g(x - 1)
```

Here we consider `f` as the main function and we can see that any path we try to take through `f` and through the terminal branch of `g` is infeasible. However, any value of `x` we chose will force us through the recursive branch of `g` and eventually will visit the terminal branch of g. Hence even when some nodes are only reachable using cycles, considering the simple paths is sufficient to find a value that will visit these nodes.

5.3 Program Execution Waypoints

In this section, we address a more general problem than the static statements coverage. Since the presence of a recursive function generally means the existence of an infinite number of execution paths through the graph, the static statements coverage we presented above cannot test all of these paths. In fact, any path that goes more than one time through the same expression will not be explicitly considered by our procedure, even though it might still end up being visited thanks to a fortuitous choice of inputs.

We introduce a special `waypoint` expression that can be used to specify waypoints across the program to dictate the path of execution. The `waypoint` expression has the semantics of an identity function, it is basically ignored in term of execution. It is tagged with an integer that indicates an ordering over all the waypoints in the program. For example:

```
waypoint#1(f(waypoint#2(x)))
```

would be a path starting from the expression `f(x)` and going through the subexpressions `x`. On the other hand, the following expression:

```
waypoint#2(f(waypoint#1(x)))
```

expresses a path starting from the expression `x` and ending at `f(x)`, which is actually infeasible in our graph representation, unless there is recursive call.

Using such notation, we can actually constraint a path to go a precise number of times in the recursive branch of a function. Consider the following program with some waypoints:

```
def sum(i) =
  if(i <= 0)
    waypoint#3(0)
  else
    waypoint#2(waypoint#1(sum(i - 1) + i))
```

The first waypoint is required to go through the recursive call, as well as the second waypoint. Finally the third waypoint is in the base case. The collected conditions from such a path form the following formula:

$$i > 0 \land i - 1 > 0 \land i - 2 \leq 0$$

which implies $i = 2$ which is the unique solution to satisfy this formula and thus calling `sum` with the value 2 will exactly visit the specified path. Note that the order of the waypoints in the recursive call is very important, switching both expressions we would actually have a trivial simple path in the graph from the first waypoint to the second waypoint resulting in a different formula that would constraint only one recursive call hence will give a solution of $i = 1$.

With waypoints, we no longer try to cover all of the expressions starting from a main function. We simply try to find one feasible path that goes exactly through all of these waypoints in the correct order. Finding one such path can seem to be of limited use. But, we could first run a preprocessing step that automatically annotate the program with waypoints, or even an enumeration technique that generate many arrangements of

```
def find(s, t, currentPath)
  forall path p from s to t
    if(isFeasible(currentPath, p))
      val found = find(t, nextWaypoint(t), augmentPath(currentPath, p))
      if(found)
        return found
  return false
```

Figure 5.4: Backtracking algorithm to find a path across multiple waypoints.

waypoints. Such techniques would be able to find a smart coverage of many execution trace and their development are somewhat orthogonal to the work done in this section.

Now, let us discuss how to find such a path. The graph has still a similar structure to the previous sections, but this time we have a list of nodes we need to visit in order. These nodes are simply the nodes corresponding to the expressions inside the waypoints. We need keep such nodes during the simplification of the graph.

We cannot independently decompose the search in finding paths from starting point to ending point for each group of two consecutive waypoints. If we do so, we might find one path from A to B and another path from B to C, both individually realizable but not realizable when merged together. We need to first search a feasible path for the first two waypoints, and then search a path compatible with the first one for the next two waypoints. If such a path does not exist, we will then need to backtrack and find another path for the previous two waypoints. Finding a path between two nodes in a graph can be done running a depth first search with the starting node as the initial node and searching for the ending node. These paths should be fairly enumerated. One possible approach is to slowly increment the maximal length of the path. The question of an efficient enumeration remains open.

Figure 5.4 shows an algorithm to do this backtracking. We enumerate all paths from a waypoint (`s`) to the following waypoint (`t`). For each such path we check whether the path is feasible, and, if so, we attempt to recursively find the path for the next pair of waypoints. At that point, either the recursive call will have successfully built a path all the way to the last waypoint and we can return it, or it will have failed and we will need to continue the loop with the next path. As a final note, this procedure does not work when two waypoints are in two parallel paths that are both executed. One should make sure that the waypoints are all along a same path of execution.

Chapter 6

The Leon Verification System

The algorithms described in this work have been implemented into the Leon verification system [59]. In this chapter, we present the integrations of these new capabilities into the existing system. Leon is written is the Scala programming language [54] and works as plug-in for the Scala compiler.

6.1 Front-End

The original Leon verification system supported a functional subset of Scala. The main supported expressions are listed below:

- Integers and Boolean literals
- Arithmetic operations over integers
- Comparisons operators over integers
- Equality between any values
- Boolean expressions
- Case classes used to encode algebraic data types with support for pattern matching
- Set and Map with basic operations

Accepted programs consist of an `object` with a collection of definitions, each of them being either a case class definition or a function definition. Note that this subset of Scala is still Turing-complete and can be used to write many interesting programs. Contracts of functions can be specified using the Scala dynamic contracts notation [53]: a `require` call at the beginning of the function indicates a precondition and an `ensuring` on the result of the function indicates a postcondition. An interesting property of this front-end is that any program analyzed by Leon is also a valid Scala program and can be compiled by the Scala compiler.

However, there is no possibility to write imperative programs: no global variable, no var construct nor while loop. Although it is possible to encode any kind of computations using the original input language, some solutions are more naturally expressed using an imperative style. The main focus of this work was to extend Leon with a verification procedure for imperative programs, however we also tried to make Leon more flexible by supporting a larger subset of Scala.

Here is a quick overview of the new supported features:

```
abstract class List
case class Cons(head: Int, tail: List) extends List
case class Nil() extends List

def size(lst: List): Int = tailSize(lst, 0)

def tailSize(lst: List, acc: Int): Int = lst match {
  case Nil() => acc
  case Cons(_, tail) => tailSize(tail, acc+1)
}
```

Figure 6.1: Size function in original leon syntax.

```
def size(lst: List[Int]): Int = {
  def rec(lst: List[Int], acc: Int): Int = lst match {
    case Nil => acc
    case _ :: tail => rec(tail, acc+1)
  }
  rec(lst, 0)
}
```

Figure 6.2: Size function in extended Leon syntax.

- Tuples with selectors and pattern matching
- List with basic operations and pattern matching
- Local variables declaration, assignments, blocks and while loops
- Unit type and literal
- Various additional operations: modulo, `instanceOf` operator
- Functional and imperative Arrays, without aliasing
- Nested functions definitions

Following the consistent syntax of Scala, almost everything is an expression, including while loops and assignments (they are expressions that return a value of type `Unit`). This means that an expression such as:

```
var i = 0
{i = i + 1; i} + {i = i + 1; i}
```

is valid and should return 3.

To illustrate the advantages brought by the new extended syntax we can consider a simple example. Figure 6.1 shows how we would write a function that computes the size of a list in the original syntax. We are required to define the List type using case classes and we need to specialize it to the `Int` type. We also need to define a top level tail recursive function to efficiently compute the size. On the other hand, Figure 6.2, which uses the extended syntax, shows an equivalent but much more natural Scala program. We believe that this new front-end can make Leon much more user friendly.

We defined two extensions to the Scala language in order to write loop invariants and the `epsilon` function. We declared the additional functions in a library so that the Scala

```
def epsilon[A](p: (A) => Boolean): A
```

Figure 6.3: Definition of `epsilon`.

```
def positiveInt(): Int = epsilon((i: Int) => i > 0)
```

Figure 6.4: Utilization of `epsilon`.

parser can still be used to parse the input. The definition of epsilon is shown in Figure 6.3. We do not need to provide any concrete implementation because our plug-in will extract this specific function and treat it particularly. In this declaration, we use Scala generics types, with the syntax `[A]`. This means the function can be used with an input of any type. An example of the use of `epsilon` in a concrete program is presented in Figure 6.4. The function `positiveInt` is defined using an `epsilon` expression and will return an integer > 0.

The utilization of a loop invariant is shown in Figure 6.5. Its declaration as a library takes advantage of some of the advanced characteristics of Scala, in particular we make use of an implicit conversion from `Unit` to an object that contains an `invariant` method. The exact declarations can be found in Figure 6.6.

6.2 Internal Representation

In Section 6.1 we describe the Scala syntax that can be used to write input programs for Leon. Now we explain the internal representation used in Leon.

The most essential part of the program that requires a discussion is the representation of the expressions. We use Scala case classes concept, which are an encoding of algebraic data types using object oriented features, to represent trees of expressions.

As a first representation after the parsing phase, we mostly mirror the Scala syntax, having one case class corresponding to each different expression parsed. It should be noted that this is very different from the internal representation in the Scala compiler. There, they decided to have a minimal number of classes to represent expressions, mainly `Apply` and `Select`. The former applies a function/method to a list of arguments while the latter selects an element from an object. This representation is made possible by the minimal and consistent syntax of Scala. This is relevant, since we need to interface Leon with the Scala compiler.

Later in the process, we use a subset of the same data structure to represent the expression during the generation and analysis of the verification conditions. This subset has not very much changed compared to the original Leon system. Here is a list of some new expressions that needed to be added:

```
var i = 0
var s = 0
(while(i < 10) {
  i = i + 1
  s = s + i
}) invariant(i <= s)
```

Figure 6.5: Utilization of a loop invariant.

```
object InvariantFunction {
  def invariant(inv: Boolean): Unit = ()
}
implicit def while2inv(u: Unit): Invariant = InvariantFunction
```

Figure 6.6: Declaration of loop invariant notation.

```
abstract class Pass {
  def apply(program: Program): Program
}
```

Figure 6.7: Interface for a `Pass`.

- Functional arrays, with store and select operations
- Tuples literals and selectors
- Unit literal

6.3 Multi-Pass Architecture

We adopted an architecture based on a series of passes. This provides great modularity and ensures better quality of the source code.

We identified a core part of the language, which corresponds roughly to the original supported sub-language. One difference is the addition of tuples that we need to support natively in our analyzer. One advantage of this pass architecture is that we were able to rely on the solid analyzer that was already used by Leon. By making only minor updates to this analyzer, we ensure that this part of the system is working safely.

Figure 6.7 shows a part of the code for the abstract `Pass` class. The `Program` type is the class representing a program in Leon. The application of passes is plugged in after the parser and before the analyzer. The passes apply successive program transformations, rewriting the full featured program into a smaller, purely functional core. We believe that this architecture makes it also easier to extend Leon with new Scala syntax, since adding something new would only require to parse it and then transform it during a new separate pass. This also makes each feature truly orthogonal to implement.

We give a list of the passes implemented in Leon so far:

- Transformation and specialization of lists to case classes.
- Transformation of imperative arrays to functional arrays using assignments.
- Transformation of epsilons to abstract functions.
- Elimination of expression with side-effect.
- Lifting of nested functions.
- Various simplification of the program.

6.4 Support for Arrays

As described previously, Leon can handle Scala arrays. Arrays are one of the fundamental data structures when one wishes to write imperative programs. However, since arrays causes several complications, we only allow a limited usage of them.

In particular, arrays being a mutable structure, they can lead to aliasing. So far, none of the features we supported could lead to an aliasing problem. This is such because we were only considering immutable structures. Since we are not able to reason about aliasing, we simply forbid any assignment of arrays that would lead to an aliasing.

An array can also be mutated while passing as an argument to a function. This is not desirable since we assume no side-effects at the level of the functions. We also reject programs containing such usage of arrays. The integration of mutable arrays into a functional program has a finer treatment in another work [52], where, for example, effect analysis with abstract interpretation is used to determined whether a destructive update is invisible. In our case, we have a much coarser analysis, and we will not accept many programs that would be perfectly safe.

For a technical reason during the transformation of mutable arrays to functional arrays, we require that any store on the array is performed on an array expression that has the form of an identifier. This is consistent with the rest of the constraints on arrays, even though one could imagine some more complicated array updates in standard Scala code.

Now we detail how exactly we detect invalid usage of arrays. Each array expression is associated with a notion of an *owner*. Each assignment of an array expression to an array variable will register an additional owner for the array variable. If a returned array expression was owned by the current function, it will lose that owner. It also is additionaly owned by an external function. An array parameter in a function is owned by another external function. If at any point, an array expression has more than one owner, then we reject the program.

6.5 Back-End Solver

We rely on the SMT solver Z3 [17]. Z3 is a very efficient solver that supports most theories of interest in programming languages. In particular, the following theories are implemented in Z3:

- Integer Linear Arithmetic
- Boolean Algebra
- Array with Extensionality
- Algebraic Data Types
- Uninterpreted Functions

Z3 has the advantage to be one of the most efficient SMT solver. This is very important since complex programs can lead to numerous verification conditions to be solved.

We can also take advantage of Z3 support for incremental reasoning. Indeed the core unrolling algorithm, as described in Section 2, incrementally adds additional constraints (and sometimes removes them).

Finally, note that we use a Scala library to interface natively with Z3 [34] and obtain access to its full power.

Chapter 7

Experiments

We are able to prove some surprisingly complex and expressive properties of both functional and imperative programs. In particular, thanks to the inductive nature of algebraic data types, we can validate many properties on such kind of programs. Most of the time, Leon is able to conclude with blazing speed. This is especially the case with counter-examples, but most of the valid properties are also proven within seconds. We have a few cases of valid properties where Leon is extremely slow, they are however part of a specific pattern and we believe there is no fundamental limitation in Leon for them, it is more of an engineering issue. We will discuss these later in this chapter.

Leon was already able to successfully verify a good number of examples. However these examples exhibits only the purely functional reasoning capabilities of Leon and could not be reused to test the new system. So we built a collection of new realistic programs, making as much use as possible of the new features, while still relying on the already existing functional core. In this chapter, we present, with a good level of detail, some of these programs and report the results of running Leon on them.

7.1 Overview

We have evaluated many properties, spanning several functions, for a total of more than 500 lines of code. Since the properties and algorithms are written in Scala, they are naturally very compact and expressive. The benchmarks were run on a computer equipped with two Intel Core i5-2500s running at 2.7 GHz and with 4.0 GB of RAM. We used Z3 version 3.2.

Tables 7.1, 7.2, 7.3 and 7.4 sum up our evaluation. The abbreviation LOC stands for lines of code. Each table corresponds to a different familiy of functions. Each line lists a generated verification condition. The first column gives the name of the function to which the verification conditions is related. The second column indicates the type of verification condition. Here is a list with a short description of each kind of verification condition:

post: the postcondition of the function. It usually corresponds to a property that verifies the correct implementation of the function.

pre: a condition that ensures that an invocation of the function meets the precondition.

array check: a check that an array access is within valid bounds.

match check: a check for the completeness of the pattern matching. It generates verification condition for statically incomplete pattern matching and can prove that the pattern matching still covers all possible cases.

Function	VC Kind	Result	Time
add	post	Valid	0.07
add	loop inv	Valid	0.04
add	loop inv	Valid	0.03
addBuggy	post	Valid	0.01
addBuggy	loop inv	Invalid	0.05
addBuggy	loop inv	Valid	0.03
mult	post	Valid	0.01
mult	loop inv	Valid	0.03
mult	loop inv	Valid	0.03
sum	post	Valid	< 0.01
sum	loop inv	Valid	0.03

Table 7.1: Verification conditions for benchmark `Arithmetic` (73 LOC).

loop inv: a loop invariant. The verification condition verifies that the invariant is valid initially, remains valid after each iteration and is valid at the exit of the loop.

The third column describes the result of the solver. It can be either Valid, Invalid or Unknown. Unknown means that the condition was too difficult to prove and the solvers would likely loop forever. Finally, the last column indicates the time needed (in seconds) to solve the formula. We use the notation < 0.01 to indicate that the time was very close to 0. Some lines are repeated multiple times, this is intended and corresponds to different verification conditions in the same function of the same kind. It can happens when, for example, there is more than one access to an array.

Overall, a good number of properties are proven valid. In particular, all match expressions and array accesses can be successfully verified. The more difficult properties to prove are the ones on arrays (see Table 7.2), but these are relatively difficult properties to prove as for example we try to prove that a sort routine indeed put the array in correct order. Also, most of the conditions are verified in less than a second, however we can see a few case where the verification is slow. We remark that the postcondition of some buggy functions is still valid, but some loop invariant inside the functions are not. The reason is that each verification condition is handled separately, and the postcondition of a function depends on the invariant inside the function. Validity can only be concluded if all of the verification conditions are actually valid.

7.2 Discussions

We now discussed the special characteristics of the examples used to collect the results reported in the previous section. Most of them revolve around an iterative algorithm written with `while` loops. First, we test some simple arithmetic computations. They are mostly toy examples, but they illustrate the use of loops while not mixing in any other programming features.

We then consider a number of examples on arrays. The Array is a typical data structure used with imperative programming and loops. We added support for arrays in Leon for this reason. Next, We consider the use of loops with recursive data types. Here, usually, it is more natural to use recursion, in particular with lists and trees, but some algorithms can also naturally be translated from a recursive version into an iterative version. Finally, we look at some examples that makes use of a constraint programming style with the `epsilon` primitive.

Function	VC Kind	Result	Time
abs	post	Valid	0.10
abs	array check	Valid	< 0.01
abs	array check	Valid	< 0.01
abs	array check	Valid	< 0.01
abs	array check	Valid	< 0.01
abs	array check	Valid	< 0.01
abs	loop inv	Unknown	
binarySearch	post	Unknown	
binarySearch	array check	Valid	0.02
binarySearch	loop inv	Unknown	
bubleSort	post	Unknown	
bubleSort	array check	Valid	0.03
bubleSort	array check	Valid	< 0.01
bubleSort	array check	Valid	< 0.01
bubleSort	array check	Valid	< 0.01
bubleSort	array check	Valid	< 0.01
bubleSort	array check	Valid	< 0.01
bubleSort	loop inv	Unknown	
bubleSort	loop inv	Unknown	
contains	prec	Valid	0.03
content	array check	Valid	< 0.01
content	loop inv	Valid	0.01
isPositive	prec	Valid	< 0.01
linearSearch	post	Valid	0.06
linearSearch	array check	Valid	< 0.01
linearSearch	loop inv	Unknown	
maxSum	post	Valid	0.03
maxSum	array check	Valid	< 0.01
maxSum	array check	Valid	< 0.01
maxSum	array check	Valid	< 0.01
maxSum	loop inv	Valid	1.79
occurs	prec	Valid	< 0.01
partitioned	array check	Valid	< 0.01
partitioned	array check	Valid	< 0.01
partitioned	loop inv	Valid	0.02
partitioned	loop inv	Valid	0.01
rec1	array check	Valid	< 0.01
rec1	prec	Valid	< 0.01
rec2	array check	Valid	< 0.01
rec2	prec	Valid	< 0.01
sorted	array check	Valid	< 0.01
sorted	array check	Valid	< 0.01
sorted	loop inv	Valid	0.02

Table 7.2: Verification conditions for benchmark `ArrayOperations` (207 LOC).

Function	VC Kind	Result	Time
append	post	Valid	0.10
append	match check	Valid	< 0.01
append	loop inv	Valid	0.13
appendBuggy	match check	Valid	< 0.01
appendEqAppendBuggy	post	Invalid	3.16
iplReverse	post	Valid	0.03
iplReverse	match check	Valid	< 0.01
iplReverse	loop inv	Valid	0.17
iplSize	post	Valid	< 0.01
iplSize	match check	Valid	< 0.01
iplSize	loop inv	Valid	0.01
listEqReverse	post	Invalid	0.42
reverse	post	Valid	0.02
reverse	match check	Valid	< 0.01
reverse	loop inv	Valid	0.02
size	post	Valid	< 0.01
size	match check	Valid	< 0.01
size	loop inv	Valid	0.01
sizeBuggy	post	Invalid	< 0.01
sizeBuggy	match check	Valid	< 0.01
sizeBuggy	loop inv	Invalid	0.01
sizeImpEqFun	post	Unknown	
zip	post	Valid	0.08
zip	match check	Valid	< 0.01
zip	match check	Valid	< 0.01
zip	loop inv	Valid	0.07

Table 7.3: Verification conditions for benchmark `ListOperations` (146 LOC).

Function	VC Kind	Result	Time
linearEquation	post	Valid	0.05
negativeNum	post	Valid	< 0.01
nonDeterministicExecution	post	Invalid	0.13
positiveNum	post	Valid	< 0.01
setSize	pre	Valid	< 0.01
setSize	post	Valid	0.01
sizeToListEq	post	Invalid	1.12
sizeToListLessEq	post	Unknown	
toList	pre	Valid	< 0.01
toListEq	pre	Invalid	1.06

Table 7.4: Verification conditions for benchmark `Constraints` (76 LOC).

```
def add(x : Int, y : Int): Int = {
  var r = x
  if(y < 0) {
    var n = y
    (while(n != 0) {
      r = r - 1
      n = n + 1
    }) invariant(r == x + y - n && 0 <= -n)
  } else {
    var n = y
    (while(n != 0) {
      r = r + 1
      n = n - 1
    }) invariant(r == x + y - n && 0 <= n)
  }
  r
} ensuring(_ == x+y)
```

Figure 7.1: Code for iterative addition.

7.2.1 Basic Loops with Arithmetic

In this first benchmark, our aim is to experiment with loops and a minimal number of features. We thus look at some arithmetic operations that can be expressed with iterative algorithms. Note that both the addition and multiplication algorithm are from a suite of benchmark presented at VSTTE 2008 [62] and solved by Dafny [40]. Let us look at one of these functions in Figure 7.1.

The function computes the addition of its two arguments by iteratively adding 1. The algorithm is relatively straightforward. The initial `if` distinguishes the cases where the second parameter is positive or negative, and then the loop will iterate over this parameter accordingly. The postcondition requires that this function indeed computes the addition of its arguments. In that case, it is easy to write the postcondition because the addition is an operation that is part of the input language, but in general this might not be the case and the postcondition could be calling some other functions.

In order for the proof to complete, we are required to give inductive invariant to the loops so that the verifier can conclude the postcondition. The invariants simply relates the current value of the returned value with the current iteration. At the end, assuming the negation of the loop condition, we can conclude the property. All the properties of this benchmark were fully verified or disproved using a counter-example. The performance is very good since all of the properties were decided in less than a second.

7.2.2 Algorithms on Arrays

In the next benchmark, we focus on verification of properties involving the array data structure. The algorithms of this section are from various sources, including the VSTTE 2010 competition, the VSTTE 2008 challenge [62] and the Calculus of Computation textbook [14].

As a fairly complicated example, let us consider the Scala program in Figure 7.2. The program implements a binary search in the input array. It returns the index of the element if found or -1 else. The algorithm makes use of a loop instead of the alternative recursive definition. The implementation is fairly standard: pick the middle element, compare it

```
def binarySearch(a: Array[Int], key: Int): Int = {
  require(a.length > 0 && sorted(a, 0, a.length - 1))
  var low = 0
  var high = a.length - 1
  var res = -1

  (while(low <= high && res == -1) {
   val i = (high + low) / 2
   val v = a(i)

   if(v == key)
     res = i

   if(v > key)
     high = i - 1
   else if(v < key)
     low = i + 1
  }) invariant(
      res >= -1 && res < a.length && 0 <= low &&
      low <= high + 1 && high >= -1 && high < a.length &&
      (if (res >= >0) a(res) == key
       else (!occurs(a, 0, low, key) &&
             !occurs(a, high + 1, a.length, key))))
  res
} ensuring(res =>
    res >= -1 && res < a.length &&
    (if(res >= 0) a(res) == key
     else !occurs(a, 0, a.length, key)))
```

Figure 7.2: Code for binary search.

```
def size(l: List) : Int = {
  var r = 0
  var l2 = l
  (while(!l2.isInstanceOf[Nil]) {
   val Cons(_, tail) = l2
   l2 = tail
   r = r+1
  }) invariant(r >= 0)
  r
} ensuring(res => res >= 0)
```

Figure 7.3: Code for size function on `List`.

to the searched element, and then continue the search in the relevant sub-array. As a precondition, we require the array to be already sorted. The `sorted` predicate is simply a function defined in Scala. This predicate verifies that the array is correctly sorted. The postcondition states that the result must be either -1 or a valid index, and if it is not -1 then it must be true that the value at this index equals the searched element. It also states that if -1 is returned then the element must not occurs in the array, using the `occurs` function also defined in this benchmark.

Finally we need to give an invariant strong enough to conclude the postcondition. We also use `occurs` to express the invariant, restricting its use to a sub-array that growth with the number of iterations. The invariant is inductive, each iteration should maintain it and at the exit of the loop it is possible to conclude that the complete array is covered by the property in the invariant. Unfortunately, as the results show, Leon is not able to conclude the validity of the invariant. It is still able to verify the array accesses. It turns out that most properties on array that require such strong invariants are too difficult to handle for Leon. It should still be noted that these are not trivial algorithms, in particular we attempted to prove the correctness of a bubble sort implementation and a linear search as well. One of the main difficulty seems to be the functions used as properties that implicitly require a universal quantifier.

7.2.3 List Operations

We revisit some List operations already presented in the original paper on Leon [59]. Since these examples were designed for the purely functional core of Leon, we rewrite those in an imperative style. We believe that properties on recursive data structures such as List are more easily provable by Leon.

Figure 7.3 shows how to write a size function over List using an imperative style. The loop test if the list variable is not the empty list, and then it use a special syntax of Scala to extract the tail from the list. Note that the Scala compiler actually translates `val Cons(_, tail) = l2` into an explicit pattern matching with only one branch (Cons, in that case). This is why some pattern matching verification conditions are generated (and are successively proven).

As the results in Table 7.3 show, most of the properties can be efficiently proven. We implemented a number of standard list operations in an imperative style, including reverse, append and zip. Reverse is especially useful because most of these list operations will build the resulting list in the reverse order, which a final call to reverse will put in the right order.

```
def nonDeterministicExecution(): Int = {
  var i = 0
  var b = epsilon((x: Boolean) => i == i)
  while(b) {
    i = i + 1
    b = epsilon((x: Boolean) => i == i)
  }
  i
} ensuring(_ <= 10)
```

Figure 7.4: Loop that can be executed any number of times.

```
def toList(set: Set[Int]): List =
  if(set == Set.empty[Int]) Nil() else {
    val elem = epsilon((x : Int) => set contains x)
    Cons(elem, toList(set -- Set[Int](elem)))
  }
def sizeToListEq(lst: List): Boolean =
  (size(toList(toSet(lst))) == size(lst)) holds
```

Figure 7.5: Accessing elements in a set with `epsilon`.

7.2.4 Constraint Programming

Here we show some possible utilizations of the `epsilon` primitive. Table 7.4 lists our results.

Figure 7.4 shows an interesting application of `epsilon` to allow a non-deterministic execution of a `while` loop. The test condition is the result of an `epsilon` expression whose predicate is always true. Thus the `epsilon` expression takes the value of `true` or `false` and the loop can stop at any iteration. We try to assert that the value of `i` at the exit of the loop is ≤ 10, which is very likely but not always true. Leon is able to find a counter-example by giving us a sequence of values that can be taken by `epsilon` to makes this postcondition false. Note that `epsilon` behaves as a non-deterministic expression here because of the variable i in scope that keeps changing its value. We described such constructions in Section 4.1.4.

Another nice use of `epsilon` can be found in Figure 7.5. Note that we used `holds` as a syntactic sugar for a postcondition that expresses the fact that the function returns true. Here we use `epsilon` as a way to select an element in a set. Leon does support sets, but none of the operations on sets can actually extract a value. Sets were mostly used in the original Leon as a way to write specifications (such as a `content` function for lists or trees). The property `sizeToListEq` tries to claim that taking the size of a list should be the same as taking the size after having transformed the list in a set and then the resulting set in a list again. This is not true because if the original list contained duplicates, those duplicates would be lost during the transformation to a set. Leon successfully finds such a counter-example.

As a last example, we can use `epsilon` as a way to solve constraints on arithmetic expressions. Figure 7.6 illustrates this ability. Here the predicate gives the following constraints:

$$2x + 3y == 10 \land x \geq 0 \land y \geq 0$$

where x and y are encoded using a tuple of integers since we only support single arguments

```
def linearEquation(): (Int, Int) = {
  val sol = epsilon((t: (Int, Int)) =>
    2*t._1 + 3*t._2 == 10 &&
    t._1 >= 0 &&
    t._2 >= 0)
  sol
} ensuring(res => res == (2, 2) || res == (5, 0))
```

Figure 7.6: Solving linear arithmetic constraints.

in `epsilon` predicates. Such constraints on integers have only two solutions: $(x = 5 \wedge y = 0) \vee (x = 2 \wedge y = 2)$. This is what we assert in the postcondition, and Leon is able to prove the validity of this function.

As we have seen, the `epsilon` construct is a very versatile tool, and being able to reason with it is important.

7.3 Extended Case Study

In this section, we present one possible application of the Leon system to program development. Most static analysis tools try to focus on proving properties about programs, however, Leon is complete for finding counter-examples, and, in our experience, is very fast at doing so. It seems one can assume the property is valid if a counter-example is not found in the first few seconds.

So, even though this method provides absolutely no correctness guarantee, it is possible to use it efficiently to develop some non trivial algorithms. The method consists in first writing the algorithm as you would do in a normal development process, then writing a function that can express some valid properties of the algorithm. There is no need for any extra annotation, simply expressing a relation that should be true assuming the correctness of the implementation. The absence of additional postconditions and invariants will probably prevent Leon from being able to prove the property, however it will be able to discover counter-examples very quickly. The programmer can then examine the counter-examples, which would be input values on which the property fails, and address the error.

If Leon seems to be looping, then it is best to assume the property is valid. At that point, it is possible to take advantage of the built-in tests generator in Leon, to obtain many input values that can be evaluated and checked for correctness.

7.3.1 Implementation of a Basic SAT Solver

In this section, we detail the thought process on using Leon as a helping tool for developers during the implementation of a small SAT solver. We then try to prove some properties of this program and are able to detect errors in the code. Once Leon no longer finds any counter-example to our properties, but is still not able to conclude their validity, we automatically generate test cases and check that they do not fail the properties.

Figure 7.7 shows the definitions for the data structure used in the implementation. The type `Formula` is used as the high level representation of formulas, with the usual logic connectives. It is intended to be used as the interface with the user, for example as the resulting tree after a parsing phase. The `ClauseList` and `VarList` are used to represent Disjunctive Normal Form (DNF) of the formulas. They are more specialized, internal data structures for the SAT solver.

We first need to be able to evaluate both representations of our formulas. Using

```
sealed abstract class Formula
case class And(f1: Formula, f2: Formula) extends Formula
case class Or(f1: Formula, f2: Formula) extends Formula
case class Not(f: Formula) extends Formula
case class Var(i: Int) extends Formula

sealed abstract class VarList
case class VarCons(head: Int, tail: VarList) extends VarList
case class VarNil() extends VarList
case class VarLit(value: Boolean) extends VarList

sealed abstract class ClauseList
case class ClauseCons(head: VarList, tail: ClauseList)
  extends ClauseList
case class ClauseNil() extends ClauseList
case class ClauseLit(value: Boolean) extends ClauseList
```

Figure 7.7: Definitions of data structure for the SAT solver.

```
def eval(formula: Formula, trueVars: Set[Int]): Boolean
def evalDnf(clauses: ClauseList, trueVars: Set[Int]): Boolean
```

Figure 7.8: Signature of evaluation functions.

these evaluation functions, we will then be able to write expressions that state that both representations are equivalent. Figure 7.8 shows the signature of these functions.

As a next step in the implementation, we wish to be able to transform a `Formula` into its DNF. We can see the signature of the function in Figure 7.9.

At that point, we have sufficient material to start writing some properties that need to be true if our implementation is correct. In particular, for all formulas f and for all assignments m of free variables in the formula, the evaluation of f with m should equal the evaluation of the DNF of f with m. This property is formalized in Figure 7.10.

While running Leon on this property, we actually were able to detect 3 different errors that prevented the property from being valid. Each time, Leon gave a counter-example that was used to guide the debugging. Then, Leon started to loop, so we assumed that the implementation was correct (or at least that there were no obvious bug) and could move on to the next part.

Now we can implement the `isSat` function in Figure 7.11. Given that the input formula is in DNF, we simply have to find one clause that is non-contradictory.

Now that the main SAT solving algorithm is implemented, we will write another property that should be true. Figure 7.12 shows one such property. It states that for all formulas f and assignments m, if f is not satisfiable then the evaluation of f at m must be false. Again, running Leon on this property should be able to find any formula and assignment that does not respect this property. Of course, finding no such counter-example is not a guarantee that the SAT algorithm is correct, but this is an acceptable starting

```
def dnf(formula: Formula): ClauseList
```

Figure 7.9: Signature of DNF function.

```
def dnfIsCorrect(formula: Formula, trueVars: Set[Int]): Boolean =
{
  val dnfFormula = dnf(formula)
  eval(formula, trueVars) == evalDnf(dnfFormula, trueVars)
} holds
```

Figure 7.10: Property asserting that `dnf` is correctly implemented.

```
def isSat(clauses: ClauseList): Boolean
```

Figure 7.11: Signature of `isSat` function.

point.

If we want a stronger correctness property, we can additionally implement a naive SAT solver that works directly on the tree data structure of the `Formula` type. This algorithm should be simple enough to be trusted and we can write a property that assert that both SAT algorithms give the same result on the same input.

As we can see, writing these specifications is relatively easy and allows us to spot most bugs. Unfortunately, Leon does not prove that these properties hold. From a user perspective, all he can observe is that Leon loops forever, trying to find a proof of validity or a counter-example but failing to do so. The reason Leon cannot prove such property is because there are many intermediate functions that are used without any specification. Given strong enough contract on each of these small utility functions, maybe Leon would be able to prove the validity of the properties. However, annotating the programs to such a degree of precision is very time consuming. This is when our automatic generation of test cases come in handy. The few specifications we have written are enough to generate a collection of input Formulas that will span a very important number of program statements. The presence of specification even mean that no manual intervention is required, because each of the input value can be automatically evaluated and the result can be automatically checked against the written specifications. Using the tests generation of Leon on our SAT solver confirms that our implementation seems correct.

```
def satIsCorrect(formula: Formula, trueVars: Set[Int]): Boolean =
{
  val dnfFormula = dnf(formula)
  if(!isSat(dnfFormula)) !eval(formula, trueVars) else true
} holds
```

Figure 7.12: Property asserting that `dnf` is correctly implemented.

Chapter 8

Conclusion

We conclude with a discussion of some of the current limitations of Leon and how it could be improved. We draw some lessons from the development of this verification procedure for imperative programming.

8.1 Limitations

As we have seen, the system supports only a subset of Scala, which means that it is not applicable to an existing software, unless a daunting rewrite is done. This, however, has the advantage to identify a well defined sub-language of Scala that can be fully and soundly analyzed. In this thesis, we try to improved on that point, allowing a much bigger subset of Scala to be used as the input language compared to before, but there are still some desirable features that are missing.

Maybe the biggest limitation in the current language is not being able to define class or structure with mutable fields. Such data structures are at the core of many non-trivial programs. One fundamental complication that comes with mutable data structures is aliasing. Aliasing occurs when two different variables point to the same structure. Changes made to the structure by accessing one of the variable will also have effects on the other one. Unfortunately, it is very difficult to track such aliasing precisely [56, 57]. Arrays are a good illustration of this limitation, although we do support some form of arrays in the input language, we prevent any assignment that would give the same array two distinct names.

Our system does not support functions as a first class citizens. In particular, this means that we cannot reason about higher order functions. Such functions are extensively used in functional programming.

Figure 8.1 shows a typical example that leads to infinite looping by Leon. The `isPositive` function is a typical example of some property we want to be able to state about arrays. It takes a `size` argument that indicates until which index (not included) the elements are positive. In particular, it is very useful when we want to have inductive invariant. We can assume as a precondition that the array is positive up to the value i, and if we are able to prove that is is positive at i after the execution of some program (usually, to body of a loop), then we can conclude that is is positive up to $i + 1$. Thus we have just complete an inductive step and if we combine it with the base case we can prove some very useful loop invariant. However, the `propertyInduct` is unfortunately not proven by Leon. The limitation seems to come from not having a bound on the evaluation of `isPositive` that is used as a property. Indeed, even a much more trivial property such as `propertyTrivial` is not proven by Leon. Note that this is a limitation of the core algorithm of Leon and not introduced by any of the transformations we presented.

```
def isPositive(a: Array[Int], size: Int): Boolean = {
  require(a.length >= 0 && size <= a.length)
  def rec(i: Int): Boolean = {
    require(i >= 0)
    if(i >= size)
      true
    else {
      if(a(i) < 0)
        false
      else
        rec(i+1)
    }
  }
  rec(0)
}

def proprertyInduct(a : Array[Int], i: Int): Boolean = {
  require(isPositive(a, i) && a.length > 0 &&
          i >= 0 && i < a.length)
  val na =
    if(a(i) < 0) a.updated(i, -a(i)) else a.updated(i, a(i))
  val ni = i + 1
  isPositive(na, ni)
} holds

def proprertyTrivial(a : Array[Int]): Boolean = {
  require(isPositive(a, a.length) && a.length > 0)
  isPositive(a, a.length - 1)
} holds
```

Figure 8.1: Property on arrays that Leon is unable to prove.

8.2 Future Work

As a future work, we are very interested in some extensions of the input language. We would like to be able to make use of global variables as well as some form of classes with mutable fields. Global variables can be integrated to the language and then mapped into the imperative extension by making each function explicitly receiving and returning the global state. Let us assume the global variables are $g_1, g_2, \ldots, g_n$. We redefine each functions in the program to take an additional n-tuple argument S representing the value of each global variable. Furthermore, we replace each use of g_i by a tuple selector $S[i]$ and each store $g_i = e$ by an assignment to the local representation of the state $S = S[e/i]$ where the element i is replaced by the new expression e. We also change the returned expression of each function to return the new global state as well. We adapt all call sites accordingly.

Modeling classes with fields is more difficult. One approach, which is the classical approach, is to use functional maps to represent them. Each field of a class becomes a map and field reads become map accesses while field writes become map updates. The receiving object being used here as the key to the map. Of course, aliasing would become an issue, so some similar restrictions to the ones used with arrays are needed. An alternative approach to the aliasing problem would be to combine with other existing tools that can provide precise pointer analysis [20, 31].

The unrolling procedure is not always powerful enough to prove all properties of interest. Leon supports some limited form of induction, and, in the future, we wish to extend the inductive reasoning capabilities of Leon in order to be able to prove stronger properties. As another proof tactic, we want to add some generalization mechanisms that would be able to replace complex expressions that are not relevant to the proof by some fresh variables. Similar techniques are used in system such as ACL2 [30].

A program can only be conclusively declared correct if one proves that it terminates in addition to meeting its contract. Our current tool is not able to ensure termination, and, in fact, assumes the program is actually terminating. The unrolling procedure at the core of our algorithm is actually only sound if the program terminates. To obtain a complete system, we plan to build on some existing techniques [43, 22, 2].

We will look into the potential integration of some limited form of higher order functions as part of our language. We hope that in some situations, higher order functions can be mapped into an uninterpreted function at the SMT level.

Proving correctness of programs still require intermediate annotations in the form of loop invariants. Our system is able to find counter-example with minimal hints, except for the actual postcondition that we wish to verify. However, in order to actually prove correctness, it will need hints in the form of inductive invariants. Some of these invariants are very difficult to find and most likely will require user intervention. However, some of these invariants are rather boring and could be automatically discovered. We plan on introducing some loop invariant generation techniques based on abstract interpretation [16, 39]. Some other simpler techniques for deriving arithmetic relations on loop iteration variables could also be used [12].

We would like to support multiple input formats. One possible approach would be to consider Scala as the intermediate language and have a preprocessor that translates any desired input formats into Scala first and then apply the Leon tool on it. Thus we would use Scala in a similar fashion as Boogie [7]. Alternatively, we could integrate support as a front-end to Leon (in the form of several parsers). We have already attempted to parse SMT-LIB2 [9] benchmarks where we represent function definitions with universal quantifiers on the parameters. Our parser supports the SMT-LIB v2 syntax, and is based on the open source tool Princess [55]. The Low Level Virtual Machine (LLVM) [36] language is an intermediate language used in compilation. It has a well formalized semantics [64] and could be suited to be translated into our internal representation, thus enabling verification of any programming language for which an LLVM front-end exists.

8.3 Conclusion

Limiting the input language to a subset of a real programming language is a matter of trade-offs. You lose expressive power but usually such power comes with complex features that will make the analysis phase to less precise. It could be possible to make a tool such as Leon accepts the complete syntax of Scala, however, in that case, features such as aliasing will complicate the analysis and we will end up losing in precision and potentially losing the soundness properties. Identifying a precisely defined sub-language can allow the system to guarantee some soundness and completeness properties. It could be worth educating the developers to limit himself to the sub-language, at the end the language is still Turing-complete and it has been shown that some non trivial programs could be written in it. Designing a programming language with verification in mind can lead to the suppression of some dangerous features that are difficult to reason about.

Formally proving properties of any decently complex software reveals to be very difficult. For one, writing the correct specifications is not easy, and is close to unmanageable when inductive invariants are needed in all loops and auxiliary functions. Our tool proved to be efficient at finding counter-examples when given a single postcondition, and our test cases generation algorithm can also provide reasonable evidences that the program verifies the contract. This approach seems much more feasible for writing complex software, and we encourage an active use of similar tool while developing a program to quickly spot errors.

In this thesis, we presented an approach to verification of imperative programs in Scala. We implemented and integrated the algorithms into the Leon verification system. Our new architecture allowed Leon to support a much richer input language as well as a completely new programming paradigm. Overall, Leon promises to be a very suitable tool to assist programmers in their tasks, given that the programmers accept to work in some sub-language of Scala. Thanks to its well defined input language, Leon is able to offer strong guarantees on its analysis, and is thus very predictable.

Bibliography

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Costa: Design and implementation of a cost and termination analyzer for java bytecode. In *of Lecture Notes in Computer Science*, pages 113–132. Springer, 2007.

[3] Allen L. Ambler, Donald I. Good, James C. Browne, Wilhelm F. Burger, Richard M. Cohen, Charles G. Hoch, and Robert E. Wells. Gypsy: A language for specification and implementation of verifiable programs. *SIGPLAN Not.*, 12(3):1–10, March 1977.

[4] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, December 1997.

[5] Andrew W. Appel. Ssa is functional programming. *SIGPLAN Not.*, 33(4):17–20, April 1998.

[6] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. pages 49–69. Springer, 2004.

[7] Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustanm. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[8] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, February 2009.

[9] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[10] Clark Barrett and Cesare Tinelli. Cvc3. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 298–302, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[12] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. Abc: algebraic bound computation for loops. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 103–118, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *Proceedings of the 11th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 317–333, Berlin, Heidelberg, 2005. Springer-Verlag.

[14] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation*. Springer, 2007.

[15] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[16] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[17] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver tools and algorithms for the construction and analysis of systems. *Tools and Algorithms for the Construction and Analysis of Systems*, April 2008.

[18] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[19] Bruno Dutertre and Leonardo de Moura. System Description: Yices 1.0.

[20] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *Proceedings of the 2006 international symposium on Software testing and analysis*, ISSTA '06, pages 133–144, New York, NY, USA, 2006. ACM.

[21] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES*, pages 193–205. ACM, 2001.

[22] Jürgen Giesl, René Thiemann, Peter Schneider-kamp, and Stephan Falke. Automated termination proofs with aprove. In *In RTA'2004: Rewriting Techniques and Applications, volume 3091 of LNCS*, pages 210–220. Springer, 2004.

[23] Mike Gordon and Hélène Collavizza. Forward with Hoare. In A. W. Roscoe, Cliff B. Jones, and Kenneth R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, History of Computing, chapter 5, pages 101–121. Springer London, London, 2010.

[24] Cordell Green. Application of theorem proving to problem solving. pages 219–239. Morgan Kaufmann, 1969.

[25] Sumit Gulwani. Synthesis from examples. *WAMBSE (Workshop on Advances in Model-Based Software Engineering) Special Issue, Infosys Labs Briefings*, 10(2), 2012.

[26] Florian Haftmann and Tobias Nipkow. A code generator framework for isabelle/hol. In *Department of Computer Science, University of Kaiserslautern*, 2007.

[27] David Hardin, Konrad Slind, Michael Whalen, and Tuan-Hung Pham. The guardol language and verification system. In *Proceedings of the 18th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12, pages 18–32, Berlin, Heidelberg, 2012. Springer-Verlag.

[28] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[29] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers, 2000.

[30] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.

[31] Etienne Kneuss. Toward interprocedural pointer and effect analysis for scala. Technical report, EPFL, 2011.

[32] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 151–164, New York, NY, USA, 2012. ACM.

[33] Viktor Kuncak, Mikael Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2010.

[34] Ali Köksal, Viktor Kuncak, and Philippe Suter. Scala to the power of z3: Integrating smt and programming. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, volume 6803 of *Lecture Notes in Computer Science*, pages 400–406. Springer Berlin / Heidelberg, 2011.

[35] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language euclid. *SIGPLAN Not.*, 12(2):1–79, February 1977.

[36] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[37] Gary T Leavens, Albert L Baker, and Clyde Ruby. Jml: A notation for detailed design. *Behavioral Specifications of Businesses and Systems*, pages 175–188, 1999.

[38] K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[39] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand, 2005.

[40] K. Rustan M. Leino and Rosemary Monahan. Dafny meets the verification benchmarks challenge. In *Proceedings of the Third international conference on Verified software: theories, tools, experiments*, VSTTE'10, pages 112–126, Berlin, Heidelberg, 2010. Springer-Verlag.

[41] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: specification inference for explicit information flow problems. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 75–86, New York, NY, USA, 2009. ACM.

[42] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, March 1971.

[43] Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In *Proceedings of the 18th international conference on Computer Aided Verification*, CAV'06, pages 401–414, Berlin, Heidelberg, 2006. Springer-Verlag.

[44] Bertrand Meyer. *Eiffel: the language.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[45] Richard Mitchell, Jim McKim, and Bertrand Meyer. *Design by contract, by example.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002.

[46] J Strother Moore. Theorem proving for verification: The early days. *Logic in Computer Science, Symposium on*, 0:283, 2010.

[47] Georg Moser and Richard Zach. The epsilon calculus (tutorial). In *CSL*, page 455, 2003.

[48] Magnus O. Myreen. *Formal verification of machine-code programs.* PhD thesis, 2009.

[49] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 333–344, New York, NY, USA, 1998. ACM.

[50] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Trans. Program. Lang. Syst.*, 11(4):517–561, October 1989.

[51] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[52] Martin Odersky. How to make destructive updates less destructive. In *IN PROC. 18TH ACM SYMP. ON PRINCIPLES OF PROGRAMMING LANGUAGES*, pages 25–36. ACM Press, 1991.

[53] Martin Odersky. Contracts for scala. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 51–57. Springer Berlin / Heidelberg, 2010.

[54] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide.* Artima Press, 2008.

[55] Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR '08, pages 274–289, Berlin, Heidelberg, 2008. Springer-Verlag.

[56] Alexandru D. Salcianu. Pointer analysis and its applications to java programs. Technical report, MIT, 2001.

[57] Alexandru D. Salcianu. Pointer analysis for java programs: Novel techniques and applications. Technical report, MIT, 2001.

[58] Guy Steele. *Common LISP. The Language.* Digital Press, 1990.

[59] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability Modulo Recursive Programs. In *Static Analysis Symposium (SAS)*, 2011.

[60] Joachim van den Berg and Bart Jacobs. The loop compiler for java and jml. pages 299–312. Springer, 2001.

[61] C. Walther and S. Schweitzer. About verifun. CADE 2003, pages 322–327, Berlin, Heidelberg, 2003. Springer.

[62] Bruce W. Weide, Murali Sitaraman, Heather K. Harton, Bruce Adcock, Paolo Bucci, Derek Bronish, Wayne D. Heym, Jason Kirschenbaum, and David Frazier. Incremental benchmarks for software verification tools and techniques. In *Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, VSTTE '08, pages 84–98, Berlin, Heidelberg, 2008. Springer-Verlag.

[63] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 349–361, New York, NY, USA, 2008. ACM.

[64] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. *SIGPLAN Not.*, 47(1):427–440, January 2012.