PROPERTY-BASED TESTING AND PROPERTIES AS TYPES: A HYBRID APPROACH TO SUPERCOMPILER VERIFICATION

JASON STEPHEN REICH

University of York
Department of Computer Science

September 2013

ABSTRACT

This thesis describes a hybrid approach to compiler verification. Property-based testing and mechanised proof are combined to support the verification of a supercompiler — a particular source-to-source program optimisation. A careful developer may use formal methods to ensure that their program code is correct to specifications. Poorly constructed compilers (and their associated machinery) can produce object code that does not have the same meaning as the source program. Therefore, to ensure the correctness of the executable program, each component of the compilation pipeline needs to be verified.

Lazy SmallCheck — a property-based testing library — is extended with support for existential qualification, functional values and a technique for displaying partial counterexamples. Lazy SmallCheck is then applied to the efficient generation of test programs for a small first-order functional language, specified using declarative statements of program validity. We extend the technique with several definitions of canonical programs to reduce the test-data space.

A supercompiler is implemented for a core higher-order language, contrasting implementations found in other publications. We also survey the techniques and themes seen in the literature on compiler proof. These surveys inform the development of an abstract verified supercompiler in a dependently-typed language. In this work, we represent correctness properties as types.

This abstract model is then adapted to integrate mechanical proof and results of property-based testing to verify a working supercompiler implementation. While more work is required to improve the framework's ease-of-use and the speed of verification, the results show that this approach to hybrid verification is feasible.

For a patient fiancée and
an understanding family.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Many PhD candidates consider themselves fortunate to have a wise, enthusiastic, challenging but kind supervisor to steer them on the path to a doctorate. I am doubly fortunate and doubly grateful to have had *two* wise, enthusiastic, challenging but kind supervisors. Colin Runciman has been stuck with me since my final-year MEng project but could still start the day with his battle cry to *"push back the frontiers of human knowledge and understanding."* Richard Paige continues to stand ready, every Friday, with a pint and the patience to listen to the week's trials. I have learned much from studying and teaching along side you both. Truly, this could not have been achieved without either of you.

The Programming Languages and Systems laboratory that I called home for four years has been populated by some of the finest researchers with whom anyone could hope to share an office. Chris Bak, Chris Poskitt, Glyn Faulkner, Ibrahim Shiyam, Jose Calderon, Matthew Naylor, Marco Perez, Michael Banks, Mustafa Aydin and Yining Zhao — thank you for your support and your company on this journey. Beyond the walls of CSE/215, I thank Ian Gray, who provided invaluable feedback on the later drafts of the thesis, those in the Enterprise Systems group (my second home) and the other members of the Large Scale Complex IT Systems initiative that funded my research.

I learned much from visits to the University of Nottingham, near the University of Nottingham and to Chalmers University of Technology. Particular thanks to Koen Claessen for agreeing to examine my thesis and to Susan Stepney, who has been a constant source of support on my thesis advisory panel. I greatly enjoyed our discussions at the viva and look forward to putting it all into practice. As they say, a thesis is a driving licence, not an epitaph.

DECLARATION

I declare that all the work in this thesis is my own, except where explicitly attributed to others. Some of the chapters are based on the following publications authored by the candidate:

- Jason S. Reich, Matthew Naylor, and Colin Runciman. Lazy generation of canonical test programs. In Andy Gill and Jurriaan Hage, editors, *Implementation and Application of Functional Languages*, volume 7257 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 2012
- Jason S. Reich, Matthew Naylor, and Colin Runciman. Advances in Lazy SmallCheck. In Ralf Hinze, editor, *Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 53–70. Springer, 2013

Part I

INTRODUCTION

1 INTRODUCTION

This chapter motivates a programme of research into hybrid methods of compiler verification. My contributions to the topic are outlined and a roadmap for the thesis is presented. There is also discussion of the notational style used by the thesis.

1.1 MOTIVATION

A compiler transforms a program in some source language into a program in a target language. A key property of a compiler should be to *preserve the semantics* of the source program in the compiler output. All compilers should meet this minimal specification. The consequences of an incorrect compiler include the introduction of (often subtle) bugs into execution behaviour, the loss of portability and impeded maintainability of both the compiler and source software.

Leroy (2009) poses the question *"Can you trust your compiler?"* Unfortunately, even for many popular compilers, the answer is not as certain as many would expect. Look at the bug tracker of your favourite compiler to find examples of vanishing loop conditions and malformed bytecode output. The language semantics may not be formally understood or, given the often infinite space of source programs, coverage provided by test suites may be too low to account for all the programs that a user may write. Some languages define their semantics as the result of compiling programs using the latest release version of some standardised compiler. This approach does not give many guarantees of software behaviour.

EXPERT TESTING Test suites for compiler correctness are often constructed by a specialist through the analysis of a language specification or observation of unexpected behaviour. This approach may be classified as *expert testing* to distinguish it from the automated testing discussed later.

When used effectively, expert testing can reveal a wide variety of programming errors. For the time invested in the implementation of testing, it can return a large improvement in confidence of software correctness. For example, a standardised suite of programs is often used to verify a compiler's correct behaviour. (Dietz, 2008; Goodenough, 1980; Partain, 1993; Wichmann and Sale, 1980) However, with compiler defect still being discovered, it would appear that additional verification methods are required.

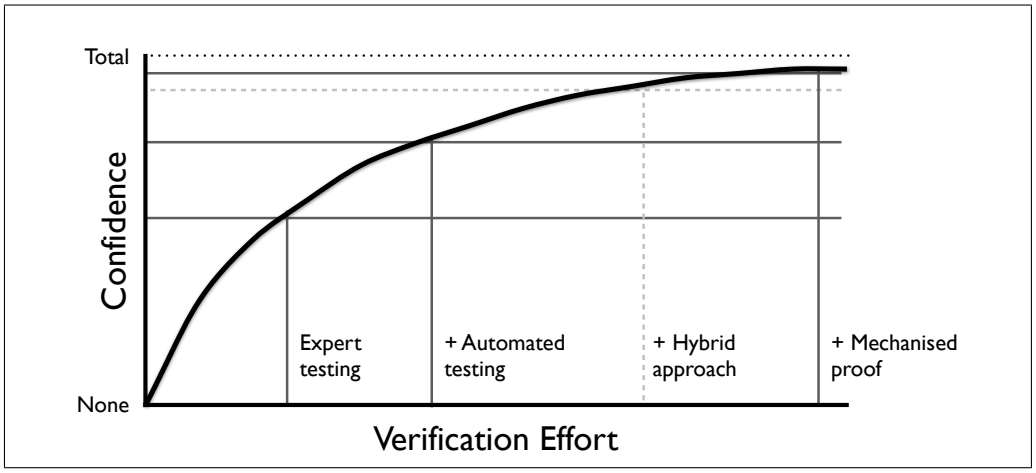

Figure 1.1: Sketch graph of confidence against verification effort.

MECHANISED PROOF In an ideal world, all compilers would be formally verified using rigorous proof techniques. The stated *"ultimate goal"* of McCarthy and Painter (1967) was to *"make it possible for a computer to check proofs that compilers are correct."* However, their seminal 1967 paper only described a hand-proof of a compiler for a small arithmetic language. It was not until Milner and Weyhrauch (1972) that the objective of mechanised checking was achieved for this small case study, as discussed in detail in Chapter 6.

Even a contemporary effort, CompCert, took *"an estimated 2 person-years of work"* (Leroy, 2009) to verify correct. It is clear that mechanised proof techniques, as they stand, are too costly to be widely accepted by the compiler community.

AUTOMATED TEST An intermediate approach, between expert-authored testing and mechanised proof, is automated testing. Examples include fuzzing (Holler, 2011) and property-based testing (Claessen and Hughes, 2000; Runciman et al., 2008). In these automated testing methods, an algorithm is used to generate tests against some correctness criteria. A developer does not have to borrow or invent examples, and can appeal to the verification algorithm as evidence of the compiler's correctness.

However, important issues need to be addressed such as the definition of compiler correctness, the method by which test programs are generated and how to ensure adequate coverage of an often infinite test-program space.

CONFIDENCE AND EFFORT Consider these three methods of verifying a compiler: expert testing, automated testing and mechanised proof. Figure 1.1 illustrates the comparative confidence we gain against effort involved with these techniques.

Assuming that all three methods use some specification of the source and target languages, any errors identified by testing would become apparent under

mechanised proof. There is still a small possibility that the proof not sound. The substantial resource costs involved in applying mechanised proof techniques are a bigger concern (Leroy, 2009).

Despite the extensive testing applied to the Glasgow Haskell Compiler, Palka et al. (2011) used a property-based testing technique to discover bugs in the optimisation phase of the compiler. The technique still does not give the near-complete confidence of mechanised proof as defects are still being discovered today.

I propose a hybrid approach that uses property-based testing to give witness to the correctness of unproven assumptions in an otherwise formally verified compiler. Hybrid methods have already been applied to verification. The *Programatica* project (Hallgren, 2003) produced tools for applying a variety of verification techniques to high-assurance systems while managing varying levels of confidence for certification. Some regions of the *CompCert* verified compiler (Leroy, 2009) use runtime assertions to support assumptions in the mechanised proof.

The application of hybrid verification has a number of advantages including; *(a)* a solid, formal foundation with clear boundaries for evidence of correctness, *(b)* partial automation of verification, *(c)* ease of reverification after requirements or implementation changes, and *(d)* modularity such that tests may be replaced with proofs as they become available.

1.2 OVERVIEW AND CONTRIBUTIONS

This thesis presents evidence supporting the assertion that:

> Combining property-based testing and mechanised proof verifies compilers with higher confidence than property-based testing alone, for less effort than mechanised proof alone.

Techniques and tools will be presented for applying property-based testing to compiler verification. A framework for combining property-based testing and mechanised proof is discussed and applied to a supercompiler implementation.

The diagram in Figure 1.2 gives an overview to the compiler development methodology proposed in this thesis. Once an appropriate definition of *correctness* has been determined with the necessary *transitive* property (see Section 7.4), the compilation process can be decomposed into verifiable modules.

Using the property-based testing techniques discussed in Chapters 2 to 4, these modules can be engineered until they test correctly and perform as desired. These modules can then gradually be formally proven in a machine-checkable logic, increasing confidence in the compiler's correctness. Once all modules have been proven correct, the correctness of the whole compiler is proven, as discussed in Chapter 7.

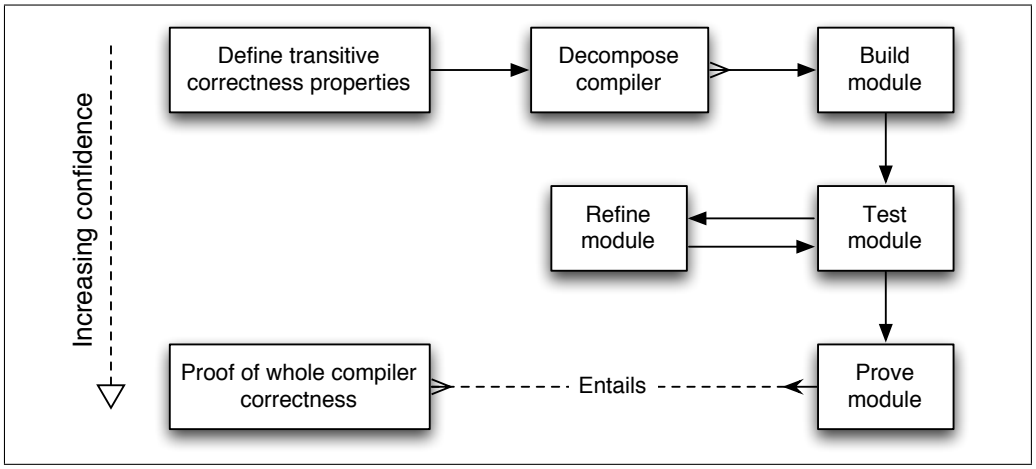

Figure 1.2: Overview of proposed methodology.

My contributions towards the topic of compiler verification are as follows:

- a new design and implementation of the Lazy SmallCheck property-based testing library to support richer properties and display partial counterexamples *(Chapter 3)*;
- an application of Lazy SmallCheck to generate functional values, exploiting the generation of partial algebraic structures *(Chapter 3)*;
- an application of Lazy SmallCheck to efficiently generate test programs from declarative definitions of validity and canonicity *(Chapter 4)*;
- a scheme for decomposing supercompilers into verifiable components *(Chapters 5 and 7)*;
- a framework for combining property-based testing and mechanised proof *(Chapter 8)*;
- a formulation and verification of a supercompiler and its correctness properties in this framework, testing assumptions made by the mechanised proof section *(Chapter 8)*.

1.3 ROADMAP

This thesis has the following structure:

- Chapter 2 introduces the key concepts of property-based testing, discusses the variety of strategies in use and reviews some of the literature on the implementation and application of the technique.

- Chapter 3 discusses improvements made to Lazy SmallCheck, a bounded exhaustive property-based testing library for Haskell. Extensions, since the original release (Runciman et al., 2008), include a richer property language and a new facility for enumerating functional values.
- Chapter 4 applies Lazy SmallCheck to the problem of enumerating test programs of functional languages. It shows how lazy predicates may be used to efficiently take a quotient of the space of test abstract syntax trees.
- Chapter 5 introduces supercompilation as a candidate for verification. A reference implementation is developed for a small, core functional language while the different design decisions are discussed with respect to the literature on the topic.
- Chapter 6 reviews key publications on compiler verification through theorem proof. It shows that program abstractions and modularity can lead to simpler proofs but proving an *"effective compiler"* correct is still time consuming.
- Chapter 7 defines a correct-by-construction abstract supercompiler in Agda, a dependently-typed language. The model supercompiler abstracts over source language and presents clear verifiable interfaces to be instantiated.
- Chapter 8 applies the framework of the abstract supercompiler to a particular supercompiler implementation, using property-based testing to check unresolved proof obligations. The work in Chapters 3 and 4 is extended to allow testing of properties over supercompiler data types beyond valid syntax trees. A method for extracting necessary tests for a supercompiler instantiation is provided.
- Chapter 9 summarises the results and conclusions of this thesis. Avenues for further work are presented.

1.4 CODE LISTINGS

In several chapters, code listings and examples are included. Any text in sans-serif indicates the code is in a host language (either Haskell or Agda). Code in the `typewriter` typeface represents the target languages or program interaction. Host language code is *Literate* (Knuth, 1984) Haskell and Agda rendered into LaTeX using *lhs2tex* and has been type-checked by the Glasgow Haskell Compiler (GHC) version 7.6.3 and Agda version 2.3.3 respectively.

Part II

PROPERTY-BASED TESTING

2 A REVIEW OF PROPERTY-BASED TESTING

This chapter introduces property-based testing, including discussion of several strategies and implementations. Particular attention is given to applications of property-based testing to compiler verification, including the verification of a small running example.

2.1 INTRODUCTION

Property-based testing is a software verification technique characterised by *(1)* the formal specification of predicates that express how a program should behave and *(2)* an automated process for finding counterexamples to those predicates. A *property-based testing library* defines the language for describing the predicates and a particular strategy for finding counterexamples.

In many cases (Claessen and Hughes, 2000; Koopman et al., 2003; Runciman et al., 2008), these predicates or *properties* are defined in the host language of the program, either as functions that result in a Boolean value or constructs of a library-specific embedded domain specific language. For example, consider the following example of a compiler from arithmetic expressions to stack-machine instructions.

```
compile :: Source → [Target]
compile (Lit n)      = [Push n]
compile (x :+ y)     = compile x ++ compile y ++ [Add]
compile (Var c)      = [Read c]
compile (Let c x y) = compile x ++ [Store c] ++ compile y
```

The syntax and semantics for these languages is defined in Figure 2.1. Environments, *Env*, map characters to integers. These represent variable bindings and register entries in the source and target languages respectively. An update function, $\Gamma\ [c \mapsto n]$, updates the variable/register c with value n in heap Γ. The *eval* function represents the direct semantics for the language whose abstract syntax is represented by *Source* and *zeros* is an environment where all variables are bound to 0. The *step* function gives the small-step operational semantics for the language whose abstract syntax is represented by *Target* and the *exec* function runs *step* for a list of *Target* instructions. The *initial* constant is the initial state of the target semantics. The *Maybe* type represents partial functions, with *Just* and *Nothing* indicating success and failure respectively.

```
  -- Variable environments
type Env = Char → Int

  -- Environment update
· [· ↦ ·] :: Char → Int → Env → Env
Γ [c ↦ n] c' | c ≡ c'     = n
             | otherwise  = Γ c'

  -- Source language abstract syntax
data Source = Lit Int                  -- Literals
            | Var Char                 -- Variable
            | Source :+ Source         -- Addition
            | Let Char Source Source   -- Local definition
  deriving Show

  -- Source language direct semantics
eval :: Source → Env → Int
eval (Lit n)      Γ = n
eval (x :+ y)     Γ = eval x Γ + eval y Γ
eval (Var c)      Γ = Γ c
eval (Let c x y) Γ = eval y (Γ [c ↦ (eval x Γ)])

  -- Environment where all variables are zero
zeros = (const 0)

  -- Target language syntax
data Target = Push Int     -- Push to stack
            | Add          -- Sum top two elements of stack
            | Store Char   -- Store top of stack
            | Read Char    -- Read to sotp of stack
  deriving Show

  -- Target language operational semantics
step :: (Env, [Int]) → Target → Maybe (Env, [Int])
step (Γ, σ)         (Push n)  = Just (Γ, n : σ)
step (Γ, m : n : σ) Add       = Just (Γ, m + n : σ)
step (Γ, n : σ)     (Store c) = Just (Γ [c ↦ n], σ)
step (Γ, σ)         (Read c)  = Just (Γ, Γ c : σ)
step (Γ, _)         _         = Nothing

exec :: (Env, [Int]) → [Target] → Maybe (Env, [Int])
exec = foldl (λstate i → state >>= flip step i) ∘ Just

  -- Initial state for target machine
initial = (zeros, [])
```

Figure 2.1: *Source* and *Target* languages for compiler example.

A desirable property of the compiler should be that all target programs compiled from source expressions should be *stack-safe*. No program should perform an *Add* instruction when the program contains less than two instructions. Neither should any program perform a *Store* instruction when the stack is empty. We define the property *prop_StackSafe* that returns *True* when a compiled *Source* expression successfully returns a result when executed.

```
prop_StackSafe :: Source → Bool
prop_StackSafe x = isJust (exec initial $ compile x)
```

Another property of the compiler should be that semantics are preserved between source expressions and compiled programs. This is defined as the property *prop_SemanticsPreserving*.

```
prop_SemanticsPreserving :: Source → Bool
prop_SemanticsPreserving x =
  Just (eval x zeros) ≡ fmap (head ∘ snd) (exec initial $ compile x)
```

In Section 2.2, we shall test these properties using two property-based testing libraries. Property-based testing offers benefits over more traditional testing methodology, such as unit testing. Developers are freed from having to invent likely failure cases. A suitable strategy will discover edge cases more reliably than human intuition. However, it may be computationally difficult to explore large input-data spaces and it is impossible to search infinite input-data spaces exhaustively.

The techniques of property-based testing are best applied to programs that do not rely on *side-effects*. Purity enables the property-based testing library to fully account for program input without having to account for implicit state. Claessen and Hughes (2002) present techniques of testing monadic code with algebraic and model specifications. However, their technique operates by making the state explicit.

2.2 STRATEGIES AND IMPLEMENTATIONS

The strategy by which a property-based testing library attempts to find counter-examples is critical. It is not simply a matter of reaching any area of the input-data space that contains failure cases. The counterexample needs to be simple and clearly presented so that it can be understood and investigated by the programmer.

2.2.1 Random selection

The seminal QuickCheck library (Claessen and Hughes, 2000) randomly samples input data to be tested against a program's properties. The probability distributions are weighted towards finite instances of infinite data types. Many different data types can be generated, including functional types. Once a counterexample is found for the property, a process known as *shrinking* can be used to find a similar but smaller counterexample.

The key assumption of the QuickCheck strategy is that should a program have a fault, there is a high probably of it appearing after testing a sample of the possible input-data. Claessen and Hughes (2000) highlight that *"random testing is most effective when the distribution of test data follows that of actual data"*. This is why QuickCheck leaves the definition of the test-data distribution to the library user who should have domain specific knowledge.

RUNNING EXAMPLE We must define a probability distribution for *Source* so that we can test our properties with the QuickCheck property-based testing library. This is done by providing an instance of the *Arbitrary* type-class, defining the *arbitrary* and *shrink* functions.

```
instance Arbitrary Source where
  arbitrary = sized arbSized
    where
      arbSized 0 = oneof
        [Lit <$> arbitrary, Var <$> arbitrary]
      arbSized n = frequency
        [(1, Lit   <$> arbitrary), (1, Var <$> arbitrary)
        , (2, (:+) <$> arbSized (n `div` 2) <*> arbSized (n `div` 2))
        , (2, Let  <$> arbitrary
                   <*> arbSized (n `div` 2) <*> arbSized (n `div` 2))]
  shrink (Lit n)      = Lit <$> shrink n
  shrink (Var v)      = Var <$> shrink v
  shrink (x :+ y)     = [x, y] ++
    [x' :+ y | x' ← shrink x] ++ [x :+ y' | y' ← shrink y]
  shrink (Let v x y) = [x, y] ++ [Let v' x y | v' ← shrink v] ++
    [Let v x' y | x' ← shrink x] ++ [Let v x y' | y' ← shrink y]
```

We use several combinators supplied by the QuickCheck library to define *arbitrary*. The *oneof* combinator creates a uniform probablity distribution between the listed elements. The *frequency* combinator, on the other hand, allows the developer to manually set the relative probabilities of the listed elements. The *sized* combinator is used to ensure that the *Source* expression

tree is finite in size by supplying a finite integer to *arbSized* which decreases on recursive calls. When the integer reaches zero only leaf syntax elements are generated. The *shrink* function provides QuickCheck with possibilities for reducing a counterexample's complexity. For our example, we simply allow it to choose subexpressions.

We can now test the properties using the *quickCheck* combinator.

```
  >>> quickCheck prop_StackSafe
+++ OK, passed 100 tests.

  >>> quickCheck prop_SemanticsPreserving
*** Failed! Falsifiable (after 67 tests and 54 shrinks):
Let 'r' (Lit 1) (Var 'a') :+ Var 'r'
```

QuickCheck finds the compiler is stack safe after 100 tests. However, after 67 tests and 54 shrinks, it finds a counterexample to the semantics preservation property. Investigation of this counterexample shows that this is due to a difference in the scoping behaviour of the source and target programs.

2.2.2 Exhaustive selection

Another approach is to systematically enumerate all possible test values up to some size or some time limit. Bounded exhaustive selection appeals to the *Small Scope hypothesis* (Jackson, 2012) that programming errors will appear for small data values. If the space is searched in order of size, the goal of QuickCheck shrinking is achieved by exhaustive selection with no further processing!

SmallCheck (Runciman et al., 2008) is an example of a size bounded property-based testing library, whereas GAST (Koopman et al., 2003) uses a time limit. Lazy SmallCheck (Runciman et al. 2008 and Chapter 3) extends the principles of SmallCheck but uses the results for partial test values to prune away regions of the test-data space, as will be discussed in Section 2.3.

RUNNING EXAMPLE To test our properties using SmallCheck (or indeed Lazy SmallCheck), we define an instance of the type-class *Serial* for *Source*, supplying an appropriate definition of *series*.

```
instance Serial Source where
  series = cons1 Lit ∪ cons1 Var ∪ cons2 (:+) ∪ cons3 Let
```

There is no need for the developer to specify probability distributions for the generators. Therefore, the *Serial* instances are often much simpler than their *Arbitrary* counterparts. The $cons_n$ combinators are used to define a series

for a construction of arity n and the $(\cup)$ operator joins together series of the same data type. These combinators are supplied by the SmallCheck and Lazy SmallCheck libraries to help the user define *Serial* instances.

We apply the *depthCheck* combinator with an integer depth bound of 4 to our properties to test with SmallCheck.

```
  >>> depthCheck 4 prop_StackSafe
Depth 4:
  Completed 6774491 test(s) without failure.

  >>> depthCheck 4 prop_SemanticsPreserving
Depth 4:
  Failed test no. 345725. Test values follow.
  Let 'a' (Lit (-1)) (Lit (-1)) :+ Var 'a'
```

Once again, the *prop_StackSafe* property appears to hold but another counterexample was found for *prop_SemanticsPreserving*. This illustrates the same compiler defect as that found by QuickCheck but with a much simpler test case. However, SmallCheck had to test considerably more values then QuickCheck to find it. No counterexample is found for *prop_SemanticsPreserving* when the depth bound is set to 3.

At this point, it is worth explaining why -1 is the *smallest* integer literal for which the property fails. Starting at a root depth bound of 4, counting the $(:+)$, *Let*, and *Lit* constructions we reach depth bound 1 for the integers. The *series* instance for integers is defined as the in-order interval $[-d\,..\,d]$ where d is the depth bound. In this case, -1 is the first value selected and it falsifies the property.

2.2.3 Comparison

Both strategies have their merits. Random testing has the potential to rapidly find useful counterexamples deep in the test-data space but is reliant on a well-tuned distribution of test-data. Systematic testing can easily be implemented generically over most test-data types.

Christiansen and Fischer (2008) attempt to abstract away from strategies by defining generators as a mapping from naturals to test-data values. A property-based testing strategy supplies natural numbers (selected randomly, exhaustively or by any method) to the mapping-generator. Property counterexamples and witnesses now have unique, simple references by which a developer can refer to specific test-data. However, this approach does impede test-data space pruning strategies such as that used by Lazy SmallCheck. Even a bounded test-data space may be computationally expensive to search for counterexamples. This is where pruning becomes essential.

Table 2.1: Values of *xs* used by Lazy SmallCheck when testing *prop_ListSizes xs*.

	Test-data	Result		Test-data	Result
(1)	$\underline{\bot}$	*Refine test-data*	*(5)*	$\bot : \bot : \underline{\bot}$	*Refine test-data*
(2)	$[\,]$	*Property satisfied*	*(6)*	$\bot : \bot : [\,]$	*Property satisfied*
(3)	$\bot : \underline{\bot}$	*Refine test-data*	*(7)*	$\bot : \bot : \bot : \underline{\bot}$	*Refine test-data*
(4)	$\bot : [\,]$	*Property satisfied*	*(8)*	$\bot : \bot : \bot : [\,]$	*Counterexample*

2.3 PRUNING THE TEST-DATA SPACE

Lazy SmallCheck, like SmallCheck, exhaustively constructs all possible values of a particular type, bounded by the depth of construction. However, it exploits Haskell's (Peyton Jones et al., 2003) *non-strict semantics* and exceptions to prune away regions of the test-data space that can be represented by a simpler example.

Lazy SmallCheck begins by testing *undefined* — $\bot$ — as the value and *refines it by need*. The demands of the test property guide the exploration of the test-data space. When evaluation of a property depends on an *undefined* component of the test-data, *exactly* that component is refined. For algebraic data types, *undefined* is refined to all possible constructions, each with *undefined* arguments. To ensure termination, when Lazy SmallCheck is run, a bound is set on the depth of possible refinements.

Consider the illustrative property *prop_ListSize*. It asserts that all lists with *Bool*-typed elements have lengths less than three.

$$
\begin{aligned}
&\mathit{prop\_ListSize} :: [\mathit{Bool}] \to \mathit{Bool} \\
&\mathit{prop\_ListSize}\ \mathit{xs} = \mathit{length}\ \mathit{xs} < 3
\end{aligned}
$$

Clearly this property is false. The Lazy SmallCheck implementation described in Chapter 3 finds the following counterexample where each occurrence of _ means *any value*.

```
 >>> test prop_ListSize
...
Depth 3:
Var 0: _:_:_:[]
```

As Lazy SmallCheck searches for this counterexample, it refines the test values bound to *xs* as shown in Table 2.1, where a bold, underlined $\underline{\bot}$ indicates the particular *undefined* component that causes refinement. Notice that the elements of the list *xs* are *never refined* as their values are *never needed* by the property. This pruning effect is the key benefit of Lazy SmallCheck over eager SmallCheck.

RUNNING EXAMPLE The same *Serial* instance is sufficient for Lazy SmallCheck as was written for SmallCheck in Section 2.2.2. To use Lazy SmallCheck, we merely swap the libraries as they have mostly compatible interfaces. For this example, we shall use the originally published implementation of Lazy SmallCheck from Runciman et al. (2008).

```
  >>> depthCheck 4 prop_StackSafe
OK, required 144541 tests at depth 4

  >>> depthCheck 4 prop_SemanticsPreserving
Counter example found:
Let 'a' (Lit (-1)) (Lit (-1)) :+ Var 'a'
```

Lazy SmallCheck returns the same results as SmallCheck but it needs less tests to achieve them. When searching for a counterexample for *prop_StackSafe*, Lazy SmallCheck only requires 2% of the tests required by SmallCheck to cover the same space. This is because particular variables and literals do not affect the stack safety of a compiled program and are, therefore, never enumerated.

2.4 APPLICATIONS AND EXPERIENCE

QUICKCHECK Since Claessen and Hughes's (2000) seminal work, QuickCheck has been applied in a variety of situations. A commercial variant, Quviq's QuickCheck for Erlang (Hughes, 2007) has, for example, found areas of ambiguity in the specification of Erlang's *'process registry'* (Hughes, 2007). Using the generated examples, the Quviq testing team formalised and documented the behaviour of a reference Erlang implementation. The Quviq tools have also been applied to the verification of automotive components (Madhavapeddy et al., 2012). Applying QuickCheck to testing against the AUTOSAR industry standard *"reduced the code size of the tests by at least an order of magnitude"* compared with those produced by outsourced experts.

CRYPTOL Cryptol (Erkök and Matthews, 2009) is a domain-specific language for describing cryptographic protocols. Its developers not only use QuickCheck to verify the implementation of Cryptol (discussed by Hughes, 2007) but also expose it to users of the language to verify their protocols (Erkök and Matthews, 2009).

EXHAUSTIVE SELECTIONS It is rarer to find specific examples of exhaustive selections, perhaps due to the commercial backing of QuickCheck through Quviq. The GAST library has been applied to the verification of reactive systems (Koopman et al., 2003) such as communication protocols.

Duke et al. (2009) applied SmallCheck to ensure algebraic properties held over operations in a reference fixed-point arithmetic library. They also compared behaviour between the reference and optimised versions of the library. The authors state that SmallCheck *"was invaluable in quickly teasing out a number of bugs"* (Duke et al., 2009).

COMPILER VERIFICATION Palka et al. (2011) describe the use of QuickCheck to generate random lambda terms for compiler testing. Their method essentially inverts the type-checker to act as a generator of well-formed and well-typed programs. Katayama (2007) takes an exhaustive approach, systematically enumerating well-typed lambda terms. Despite the difference in strategy used to select test-data terms, both approaches directly construct only well-typed terms.

In a case study, Pike et al. (2012) investigates several techniques for producing a high-assurance compiler for an embedded domain specific language. The authors reports that QuickCheck testing *"is so easy to implement and so effective that no EDSL compiler should be without it."* They also choose to generate type-correct programs directly but briefly mention the alternative of generating all possible abstract syntax trees and filtering away any that are unsuitable for compiler testing. We shall adopt this second approach in Chapter 4.

2.5 SUMMARY

Property-based testing is a lightweight method of verifying software, using an algorithm to search for counterexamples to specifications written as properties. Program properties are often defined in the host language as functions returning a Boolean value.

We have discussed two methods for selecting the test-data with which these properties are instantiated — random selection and bounded exhaustive selection. Random selection has the opportunity to test a wide variety of complex test-data but exhaustive selection gives guarantees of coverage. In Chapter 3, we will investigate another benefit of exhaustive selection — *existential quantification*. We shall also look at exploiting the lazy pruning strategy to produce counterexamples that are more concise and focused.

One of the arguments against the use of bounded exhaustive selection is the sheer volume of test-data terms to be tested. The strategy used by Lazy SmallCheck prunes this space effectively, exploiting the non-strict semantics of the host language. In Chapter 4, we will use this functionality to produce a collection of test programs suitable for compiler testing, using declarative definitions of desirable features.

3 ADVANCES IN LAZY SMALLCHECK

This chapter presents improvements to the Lazy SmallCheck property-based testing library. Users can now test properties that quantify over first-order functional values and nest universal and existential quantifiers in properties. When a property fails, Lazy SmallCheck now accurately expresses the partiality of the counterexample. These improvements are demonstrated through several practical examples. The work is used to support the property-based testing component of the hybrid verification.

3.1 INTRODUCTION

As discussed in Chapter 2, property-based testing is a lightweight approach to verification where expected or conjectured program properties are often defined in the source programming language. For example, consider the following conjectured property that in Haskell every function with a list of Boolean values as an argument, and a single Boolean value as result, can be expressed as a *foldr* application.

```
prop_ReduceFold :: ([Bool] → Bool) → Property
prop_ReduceFold r = exists $ λf z →
  forAll $ λxs → r xs ≡ foldr f z xs
```

Like all other properties used as examples in this chapter, this property does not hold; our goal is to find a counterexample. When this property is tested using our advanced version of *Lazy SmallCheck*, a small counterexample is found for r.

```
 >>> test prop_ReduceFold
...
Depth 2:
Var 0: { [] -> False
       ; _:[] -> False
       ; _:_:_ -> True }
```

The counterexample is a function that tests for a multi-item list. It is expressed in the style of Haskell's case-expression syntax. Several new features of Lazy SmallCheck are demonstrated by this example. (1) Two of the quantified variables, r and f, are *functional values*. (2) An *existential quantifier* is used in

the property definition. (3) The counterexample found for r is *concise* and understandable.

Previous property-based testing libraries struggle with such a property. The QuickCheck (Claessen and Hughes, 2000) library does not support existentials as random testing *'would rarely give useful information about an existential property: often there is a unique witness and it is most unlikely to be selected at random'* (Runciman et al., 2008). QuickCheck also requires that functional values be wrapped in a *modifier* (Claessen, 2012) for shrinking and showing purposes.

The original Lazy SmallCheck (Runciman et al., 2008) supports neither existentials nor functional values. SmallCheck (Runciman et al., 2008) supports all the necessary features of the property. However, it takes longer to produce a more complicated looking counterexample. This is because SmallCheck enumerates only fully defined test data and shows functions only in part, by systematically enumerating small arguments and corresponding results.

3.1.1 Contributions

This chapter discusses the design, implementation[1] and use of new features in Lazy SmallCheck. We present several contributions:

- An algorithm for checking properties that may contain universal and existential quantifiers in a Lazy SmallCheck-style testing library.
- A method of lazily generating and displaying *functional values*, enabling the testing of higher-order properties.
- An evaluation of these additions with respect to *functionality and run-time performance*.

3.1.2 Roadmap

Section 3.2 demonstrates the new features of the Lazy SmallCheck through several examples. Section 3.3 describes architectural changes that enable these new features. Section 3.4 presents the formulation of functional values. Section 3.5 evaluates the new Lazy SmallCheck in comparison to other Haskell property-based testing libraries. Section 3.6 summarises our findings.

1 Source code available at http://github.com/UoYCS-plasma/LazySmallCheck2012.

3.2 NEW FEATURES IN ACTION

The following examples further illustrate the new features in Lazy SmallCheck. The first generates *functional values* and displays *partial counterexamples*. The second shows the benefits of generating *small, partial functional values*. The final example demonstrates *existential quantification*.

3.2.1 Left and right folds

Let us look for a counterexample of another conjectured property. This property states that *foldl1 f* gives the same result as *foldr1 f* for non-empty list arguments with natural numbers as the element type.

```
prop_foldlr1 :: (Peano → Peano → Peano) → [Peano] → Property
prop_foldlr1 f xs = (¬ ∘ null) xs ⟹ foldl1 f xs ≡ foldr1 f xs
```

As in the original Lazy SmallCheck (Runciman et al., 2008), testing this property requires a *Serial* instance for the *Peano* data type. Additionally, an *Argument* instance must be defined so that Lazy SmallCheck can produce functional values with *Peano* arguments. We have defined a *Template Haskell* function (Sheard and Peyton Jones, 2002) — *deriveArgument* — that derives a suitable *Argument* instance automatically. Section 3.4.2 discusses this in more detail.

```
data Peano = Zero | Succ Peano
  deriving (Eq, Ord, Show, Data, Typeable)
instance Serial Peano where
  series = cons0 Zero <|> cons1 Succ
deriveArgument ''Peano
```

Lazy SmallCheck finds a counterexample at depth 3. The function *f* returns *Succ Zero* if its input is *Zero* and returns *Zero* in all other cases. The list *xs* is of length three where the last element is *Zero*.

```
 >>> test prop_foldlr1
Depth 3:
...
Var 0: { _ -> { Zero -> Succ _
              ; Succ _ -> Zero } }
Var 1: _:_:Zero:[]
```

3.2.2 Generating predicates

Our next example is based on *prop_PredicateStrings* from Claessen (2012).

```
prop_PredStrings :: (String → Bool) → Property
prop_PredStrings p = p "David" ⟹ p "Tony"
```

Lazy SmallCheck finds as a counterexample the function p that returns *True* when the second character in its argument is `'a'` and *False* when any other character occurs in the second position. The function is *undefined* for strings of length less than two.

```
 >>> test prop_PredStrings
...
Depth 4:
Var 0: { _:'a':_ -> True
       ; _:_:_ -> False }
```

Why is this the first counterexample found? We might expect a function that distinguishes an initial 'L' from an initial 'S'. As the depth-bound for testing increases, the extent to which the spines of list arguments can be refined increases. But also the range of character values used in refinements increases and the smallest non-empty range contains just 'a'.

QuickCheck also finds counterexamples for this property but the functions are stricter. They test equality with one of whole strings `"Lazy SmallCheck"` or `"SmallCheck"`.

3.2.3 Prefix of a list

This example is taken from Runciman et al. (2008). We assert that a (flawed) definition of *isPrefix* satisfies a soundness specification of the function.

```
isPrefix :: Eq a ⇒ [a] → [a] → Bool
isPrefix []       _        = True
isPrefix (x : xs) (y : ys) = x ≡ y ∨ isPrefix xs ys
isPrefix _        _        = False

prop_isPrefixSound xs ys = isPrefix (xs :: [Peano]) ys ⟹
  (exists $ λxs' → xs ++ xs' ≡ ys)
```

In Runciman et al. (2008), this property could only be checked by SmallCheck as Lazy SmallCheck did not support existential properties. Running it through the new Lazy SmallCheck gives another concise counterexample: if the first argument of *isPrefix* is a multi-item list with first element `Zero`, and the second argument is `[Zero]`; then *isPrefix* incorrectly returns `True`.

```
 >>> test prop_isPrefixSound
...
Depth 2:
Var 0: Zero:_:_
Var 1: Zero:[]
```

A smallest counterexample with both *xs* and *ys* non-empty suggests an error in the second equation defining *isPrefix*. Indeed, a disjunction has been used in place of a conjunction.

3.2.4 Quantifying over larger domains

Users of our advanced Lazy SmallCheck must be aware that, much like SmallCheck, by default any nested quantifiers are tested to the same depth as the root universal quantifier. In some cases, however, the witness for an existential may occur at deeper level than the deepest value in the domain of the root universal quantifier. Lazy SmallCheck, therefore, returns a false negative for the property. In these cases, the framework user will need to increase depth of the domain over which the existential quantifies.

Consider two formulations of *prop_apex* and *prop_apex′* from the example in Runciman et al. (2008). To satisfy the properties Lazy SmallCheck must find a value for *zs* such that it is the concatenation of any *xs* and *ys*.

$$
\begin{aligned}
&\mathit{prop\_apex}, \mathit{prop\_apex'} :: [\mathit{Bool}] \to [\mathit{Bool}] \to \mathit{Property} \\
&\mathit{prop\_apex}\ \ \mathit{xs}\ \mathit{ys} = \mathit{exists} \qquad\qquad\qquad\quad \$\, \lambda \mathit{zs} \to \mathit{xs} +\!\!+ \mathit{ys} \equiv \mathit{zs} \\
&\mathit{prop\_apex'}\ \mathit{xs}\ \mathit{ys} = \mathit{existsDeeperBy}\ (*2)\, \$\, \lambda \mathit{zs} \to \mathit{xs} +\!\!+ \mathit{ys} \equiv \mathit{zs}
\end{aligned}
$$

The *prop_apex* formulation of the property is never unsatisfiable at any depth, as witnessed by the following executions.

```
 >>> depthCheck 1 prop_apex
Var 0: [False]
Var 1: _:[]
 >>> depthCheck 2 prop_apex
Var 0: [False]
Var 1: False:_:[]
 >>> depthCheck 3 prop_apex
Var 0: [False]
Var 1: False:False:_:[]
```

This is because the satisfying witness for list *zs* will need to be of length equal to the sum of the lengths of the deepest instantiations of *xs* and *ys*. The property *prop_apex′* accounts for this using the *existsDeeperBy* combinator, doubling the depth of the domain for *zs*.

3.3 IMPLEMENTATION OF NEW LAZY SMALLCHECK

This section describes in detail how *new Lazy SmallCheck* achieves the process outlined in Section 2.3. We shall return to the *prop_ListSize* example discussed in Section 2.3 to illustrate the data types used in the implementation.

In places, instead of the actual definitions used in the implementation, we give simpler versions that are less efficient but easier to read. These differences will be summarised in Section 3.3.5.

ABSTRACTIONS We will make extensive use of the *Functor*, *Applicative* and *Alternative* type-classes. All are defined in Figure 3.1. Functors are *containers* with an associated *fmap* operation that applies functions to each contained element. Lists, for example, are functors under the *map* function.

Applicative functors (McBride and Paterson, 2008) extend this by viewing containers as *contexts* from which values may be obtained. Any ordinary value can be wrapped up in a context using *pure*. A function-in-context can be applied to a value-in-context using the $(<*>)$ operator. Returning to the lists example, *pure* places the value into a singleton list and *fs* <*> *xs* applies every function in the collection *fs* to every argument in collection *xs* to obtain a collection of results.

Alternative functors are an extension of applicative functors by the addition of an *empty* container and an operation, $(<|>)$, to merge containers. For lists, *empty* is the empty list and $(<|>)$ is list concatenation.

3.3.1 Partial values

REFINEMENT EXCEPTIONS As highlighted in Section 2.3, the test-data space includes partial values that are refined by need during the search for a counter-example. When the value of an *undefined* is needed, an exception tagged with the location of the *undefined* is raised and caught by the testing algorithm. The implementation uses GHC's *user-defined exceptions*. (Marlow, 2006) Lazy SmallCheck's *refinement exceptions* are defined in Figure 3.2.

The *Location* information uniquely identifies the component of a partial test-data value that is needed by a property under test. The *Path* in a *Location* gives directions from the root of a binary-tree representation to some specific subtree. The *Nesting* in a *Location* is akin to a *de Bruijn (1972) level*: it identifies the quantifier for the test-data variable that needs refining.

PARTIAL VALUES FUNCTOR A functor of *Partial* values is defined in Figure 3.3. The only method of accessing the value inside the *Partial* functor is through *runPartial*. It forces the result of a computation using partial values and catches any refinement exception that may be raised.

```
class Functor f where
  fmap :: (a → b) → f a → f b

infixl 3 <|>
infixl 4 <*>, <$>

(<$>) = fmap

class Functor f ⇒ Applicative f where
  pure  :: a → f a
  (<*>) :: f (a → b) → f a → f b

class Applicative f ⇒ Alternative f where
  empty :: f a
  (<|>) :: f a → f a → f a
```

Figure 3.1: Definition of *Functor*, *Applicative* and *Alternative* type-classes.

```
type Location = (Nesting, Path)
type Nesting = Int
type Path = [Bool]

data Refine = RefineAt Location deriving (Show, Typeable)
instance Exception Refine
```

Figure 3.2: Definition of *Location* carrying exceptions.

```
newtype Partial a = Partial {unsafePartial :: a}

instance Functor Partial where
  fmap f (Partial x) = Partial (f x)

instance Applicative Partial where
  pure = Partial
  Partial f <*> Partial x = Partial $ f x

runPartial :: (NFData a) ⇒ Partial a → Either Refine a
runPartial value = unsafePerformIO $
  (Right <$> evaluate (force (unsafePartial value)))
    `catch` (return ∘ Left)

refineAt :: Location → Partial a
refineAt = Partial ∘ throw ∘ RefineAt
```

Figure 3.3: Definition of the *Partial* values functor.

A *Show* instance is defined so that *Partial* values can be printed. The definition is omitted here but it follows the *'Chasing Bottoms'* (Danielsson and Jansson, 2004) technique. This is what allows the display of *wildcard patterns* in counter-examples.

RUNNING EXAMPLE Consider the third value, $\bot : \bot$, tested in Table 2.1 from Section 2.3. Here is its simplified representation and the results of two small computations using it.

```
 >>> let step3 =
    (:) <$> refineAt (0, [False, True])
        <*> refineAt (0, [True])  :: Partial [a]

 >>> runPartial (prop_ListSize <$> step3)
Left (RefineAt (0,[True]))

 >>> print (step3 :: Partial [Bool])
_:_
```

The *undefined* arguments of the list-*cons* are uniquely tagged by locations. The result of applying *prop_ListSize* shows that the second argument is needed. Pretty-printing this partial value hides the complexity underneath.

3.3.2 Test-value terms

The representation of a test-value term contains *tValue*, the information needed to obtain a partial test-data value, and *tRefine*, its possible refinements. The *Term* data type is defined in Figure 3.4.

The *Applicative* instance for terms shows how: (1) the *Path* component of a location is extended through the argument of *tValue* and (2) the *tRefine* uses this information to pass the rest of the path to the relevant subterm.

The *mergeTerms* function demonstrates how a collection of terms can be turned into a single *undefined* value paired with the ability to obtain the collection when required. This is key to the strategy illustrated in Section 2.3.

TEST-VALUE ENVIRONMENTS After test data is generated but before a property is applied to it, a pretty-printed representation of the partial value is recorded. The benefit of this technique is that we need not record a pretty-printing that could be obtained from the *final test-value* derived from the term. This will be especially useful for the display of functional values in Section 3.4. The *test-value environments* type is shown in Figure 3.5. We omit *AlignedString* but it follows established pretty-printing techniques, such as that by Hughes (1995).

```
data Term a = Term { tValue  :: (Location → TVE (Partial a))
                   , tRefine ::  (Path     → [Term a]) }
instance Functor Term where
  fmap f (Term v es) = Term ((fmap ∘ fmap ∘ fmap $ f) v)
                            ((fmap ∘ fmap ∘ fmap $ f) es)
instance Applicative Term where
  pure x = Term (pure ∘ pure ∘ pure $ x) (pure [])
  fs <*> xs = Term
    (λ(n, ps) → (<*>) <$> tValue fs (n, ps ++ [False])
                      <*> tValue xs (n, ps ++ [True]))
    (λ(p : ps) → if p then fmap (fs <*>) (tRefine xs ps)
                      else fmap (<*> xs) (tRefine fs ps))
mergeTerms :: [Term a] → Term a
mergeTerms xs = Term (TVE [string "_"] ∘ refineAt) (const xs)
```

Figure 3.4: Definition of test-value terms and a merging operation.

```
data TVE a = TVE { tveEnv :: TVInfo, tveVal :: a }
type TVInfo = [AlignedString]
instance Functor TVE where
  fmap f (TVE ctx val) = TVE ctx (f val)
instance Applicative TVE where
  pure = TVE []
  TVE ctx0 f <*> TVE ctx1 x = TVE (ctx0 ++ ctx1) (f x)
```

Figure 3.5: Definition of test-value environments.

```
type Depth = Int
newtype Series a = Series { runSeries :: Depth → [Term a] }
instance Applicative Series where
  pure = Series ∘ pure ∘ pure ∘ pure
  Series fs <*> Series xs = Series $ λd →
    [f <*> mergeTerms x | d > 0, f ← fs d
                        , let x = xs (d − 1), (¬ ∘ null) x]
instance Alternative Series where
  empty = Series $ pure []
  Series xs <|> Series ys = Series $ (++) <$> xs <*> ys
```

Figure 3.6: Definition of *Series* generators.

```
class (Data a, Typeable a) ⇒ Serial a where
  series :: Series a
  seriesWithEnv :: Series a
  seriesWithEnv = Series $ fmap storeShow <$> runSeries series
storeShow :: (Data a, Typeable a) ⇒ Term a → Term a
storeShow (Term v es) = Term
  ((fmap $ λ(TVE _ x) → TVE [string $ show x] x) v)
  (fmap storeShow <$> es)
```

Figure 3.7: Definition of the *Serial* type-class.

3.3.3 Test-value series generators

SERIES FUNCTOR Properties are tested against depth-bounded test-data terms. The Lazy SmallCheck library defines instances for the test-data *Series* functor that implicitly enforces depth-bounding and the introduction of partial test-data values. These definitions are in Figure 3.6. As with the original Lazy SmallCheck, a depth-cost is only introduced on the right-hand side of binary applications so that each child of a constructor is bounded by the same depth.

RUNNING EXAMPLE The following are definitions for depth-bounded values of Booleans, polymorphic lists and Boolean lists.

```
>>> let boolSeries = pure False <|> pure True
>>> let listSeries elem = pure []
                <|> (:) <$> elem  <*> listSeries elem
>>> let listBoolSeries = listSeries boolSeries
```

SERIAL CLASS A class of *Serial* types is defined in Figure 3.7. Lazy SmallCheck uses *Serial* instances to automatically generate test values for argument variables in properties. Using the generic *Series* operators of Figure 3.6, a family of $cons_n$ combinators can be defined exactly as described by Runciman et al. (2008).

RUNNING EXAMPLE AGAIN The library defines the series generators for many data types. The *Serial* instances for *Bool* and lists are as below. Notice that we no longer explicitly define how the arguments of list-*cons* are instantiated. It is automatically handled by the type system.

```
instance Serial Bool where
  series = cons0 False <|> cons0 True
instance Serial a ⇒ Serial [a] where
  series = cons0 [] <|> cons2 (:)
```

```
data Property = Lift Bool | Not Property
              | And Property Property   | Implies Property Property
              | ForAll (Series Property) | Exists (Series Property)
```

Figure 3.8: The underlying representation of the *Property* DSL.

3.3.4 Properties and their refutation

PROPERTIES The *Property* data type in Figure 3.8 defines the abstract syntax of a domain-specific language. It includes standard Boolean operators. Crucially, it also provides a representation of universal and existential quantifiers that supports searches for counterexamples and witnesses.

Though not defined here, smart wrappers are provided for all six *Property* constructions. These automatically lift *Bool*-typed expressions to *Property* and instantiate free variables in properties with appropriate series from *Serial* instances.

REFUTATION OF PROPERTIES The *depthCheck* function takes as arguments an integer depth-bound and a *Testable* property that may contain free variables of types of any *Serial* type. The *counterexample* and *refute* functions given in Figure 3.9 search for a failing example.

A key point to observe is that *refute* recurses when it encounters a nested quantification. All refinement requests must therefore be tagged with the *Nesting level* for the associated quantifier. The *RefineAt* information can then be passed onto the relevant *tRefine* function. Those refined terms are then prepended onto the list of terms left to test.

3.3.5 Differences between versions of Lazy SmallCheck

The main differences between the new Lazy SmallCheck and the original Lazy SmallCheck described in (Runciman et al., 2008) are as follows. In the new implementation:

- Terms are generated directly using continuations. Previously they were deserialized from a generic description.
- Terms can carry a *test-value environment* enabling the display of test-data types (such as functions) that cannot be directly pretty-printed.
- The testing algorithm calls itself recursively, refining information about enclosing quantifiers.

```
counterexample :: Depth → Series Property → Maybe TVInfo
counterexample d xs = either ⊥ id $ refute 0 d xs

refute :: Nesting → Depth → Series Property →
          Either Refine (Maybe TVInfo)
refute n d xs = terms (runSeries xs d)
  where
    terms :: [Term Property] → Either Refine (Maybe TVInfo)
    terms [] = Right Nothing
    terms (Term v es : ts)
            = case (join ∘ runPartial ∘ fmap prop) <$> v (n, []) of
      TVE _    (Left (RefineAt (m, ps)))
                  | m ≡ n     → terms $ es ps ++ ts
                  | otherwise → Left  $ RefineAt (m, ps)
      TVE info (Right False)
                  → Right $ Just info
      TVE _    (Right True)
                  → terms $ ts

    prop :: Property → Either Refine Bool
    prop (Lift    v)   = pure v
    prop (Not     p)   = ¬          <$> prop p
    prop (And     p q) = (∧)        <$> prop p <*> prop q
    prop (Implies p q) = ( ⟹ )     <$> prop p <*> prop q
    prop (ForAll  xs)  = isNothing <$> refute (succ n) d xs
    prop (Exists  xs)
      = isJust   <$> refute (succ n) (succ d) (fmap Not xs)
```

Figure 3.9: Definition of the refutation algorithm.

The main differences between the real implementation of the new Lazy SmallCheck and the slightly simplified variant described in this chapter are as follows. In the real implementation:

- The *Path* data type is a *difference list* to optimise the list-*snoc* operation.
- Terms representing total and partial values are distinguished to optimise performance and to allow the use of existing *Show* instances for total terms.
- Terms representing partial values record the total number of potential refined values they represent up to the depth bound. The refutation algorithm counts the actual number of refinements performed. *(This is useful for performance measurements and comparison with other approaches.)*

3.4 IMPLEMENTING FUNCTIONAL VALUES

The key to generating functional values is the ability to represent them as tries, also known as prefix trees. New Lazy SmallCheck supports the derivation of appropriate tries for given argument types, and the conversion of tries into functions to be used as test values.

The use of test-value environments allows a trie to be pretty-printed *before* it is converted into a Haskell function. This removes the need for the kind of modifier used by Claessen (2012).

3.4.1 Trie representations of functions

We define a generic trie data type in Figure 3.10. It is expressed as a two-level, mutually recursive GADT. Level one describes functions that either ignore their argument — *Wild*, or perform a case inspection of it — *Case*.

Level two represents details of a case inspection. The *Valu* construction occurs when the argument is of unit type and therefore returns the single result. The *Sum* construction represents functions with a tagged union as argument type, performing further inspection on their constituent types. The *Prod* construction represents functions with arguments of a product type, producing a trie that first inspects the left component of the product, then the right to return a value.

A construction *Natu vs v* represents a function with a natural number argument. If an argument *n* is less than the length of *vs*, the value of *vs* !! *n* is returned. Otherwise *v* is returned as default. The *Cast* construction is used in all other cases. We shall say more about it in Section 3.4.2. The function *applyT* converts a trie into a Haskell function.

```
type (:→:) = Level1

data Level1 k v where
  Wild :: v → Level1 k v
  Case :: Level2 k v → Level1 k v

data Level2 k v where
  Valu :: v                                  → Level2 () v
  Sum  :: Level2 j v → Level2 k v            → Level2 (Either j k) v
  Prod :: Level2 j (Level2 k v)              → Level2 (j, k) v
  Natu :: [v] → v                            → Level2 Nat v
  Cast :: Argument k ⇒ Level1 (Base k) v     → Level2 (BaseCast k) v

applyT :: (k :→: v) → k → v
applyT (Wild v) = const v
applyT (Case t) = applyL2 t

applyL2 :: Level2 k v → k → v
applyL2 (Valu v)    _           = v
applyL2 (Sum t _)   (Left k)    = t `applyL2` k
applyL2 (Sum _ t)   (Right k)   = t `applyL2` k
applyL2 (Prod t)    (j, k)      = t `applyL2` j `applyL2` k
applyL2 (Natu m d)  (Nat k)     = foldr const d $ drop k m
applyL2 (Cast t)    (BaseCast k) = t `applyT` k
```

Figure 3.10: Definition of the two-level trie data structure.

```
class (SerialL2 (Base k), Typeable k, Data k) ⇒ Argument k where
  type Base k
  toBase   :: k → Base k
  fromBase :: Base k → k
data BaseCast a = BaseCast {forceBase :: Base a}
toBaseCast :: Argument k ⇒ k → BaseCast k
toBaseCast = BaseCast ∘ toBase
fromBaseCast :: Argument k ⇒ BaseCast k → k
fromBaseCast = fromBase ∘ forceBase
```

Figure 3.11: Definition of the *Argument* type-class.

```
instance Argument Peano where
  type Base Peano = Either () (BaseCast Peano)
  toBase Zero          = Left ()
  toBase (Succ n)      = Right $ toBaseCast n
  fromBase (Left _)    = Zero
  fromBase (Right n)   = Succ $ fromBaseCast n
```

Figure 3.12: The *Argument* instance for *Peano*.

3.4.2 Custom data types for functional value arguments

The *Argument* class is defined in Figure 3.11. Users supply an instance *Argument t* to enable generated functional test values with an argument of type *t*. Each instance defines a *base type representation* and an *isomorphism* between the argument type and the base type. This is a variation of the generic trie technique used by Hinze (2000). The *Cast* construction of the trie data type performs the necessary type conversions using the *Argument* instances.

The *BaseCast* functor is used at recursive points to prevent infinite representations of recursive data types. It is a type-level thunk indicating that an arbitrary type can be translated into a *Base* type. For example, Figure 3.12 shows the *Argument Peano* instance. The Template Haskell function *deriveArgument* automatically produces *Argument* instances for any Haskell 98 type.

3.4.3 Serial instances of functional values

Functional values have been reified through the trie data type, so we first need to define series of types. The *Serial* instances are defined in Figure 3.13. A

```
seriesT :: (SerialL2 k) ⇒ Series v → Series (k :→: v)
seriesT srs = (Wild <$>^ srs) <|> (Case <$> seriesL2 srs)

class SerialL2 k where
  seriesL2 :: Series v → Series (Level2 k v)

instance SerialL2 () where
  seriesL2 srs = Valu <$>^ srs

instance (SerialL2 j, SerialL2 k) ⇒ SerialL2 (Either j k) where
  seriesL2 srs = Sum <$>^ seriesL2 srs <*>^ seriesL2 srs

instance (SerialL2 j, SerialL2 k) ⇒ SerialL2 (j, k) where
  seriesL2 srs = Prod <$>^ seriesL2 (seriesL2 srs)

instance SerialL2 Nat where
  seriesL2 srs = Natu <$>^ fullSizeList srs <*>^ srs

instance Argument k ⇒ SerialL2 (BaseCast k) where
  seriesL2 srs = Cast <$>^ seriesT srs
```

Figure 3.13: Definition of *Series* generators for tries and functions.

special type-class *SerialL2* is defined. It represents types that can be represented as trie constructions. The applicative operators with a *carret suffix* introduce *no depth cost*, as opposed to those defined in Section 3.3.3. These specialist operators have been carefully placed to give a natural depth metric for functions while keeping the series finite.

Using these definitions, a *Serial* instance for functional values is defined. The default definition of *seriesWithEnv* is overridden to store the pretty-printed form of the trie before it is converted into a Haskell function.

3.5 DISCUSSION AND RELATED WORK

A feature comparison of several Haskell property-based testing libraries can be found in Table 3.1. The test-space exploration strategy is the main distinction between the QuickCheck library and SmallCheck family of libraries. QuickCheck assumes that test data detecting a failure is likely within some probability distribution. SmallCheck, on the other hand, appeals to the *Small Scope hypothesis* (Jackson, 2012) — programming errors are likely to appear for small test data.

Table 3.1: Comparison of property-based testing library features.

Feature	QuickCheck	SmallCheck	Original LSC	New LSC
Test strategy	Random	Bounded exhaustive	Bounded exhaustive	Bounded exhaustive
Test-space pruning	N/A	N/A	Lazy generation	Lazy generation
Minimal result	Shrinking	Natural	Natural	Natural
Functional values	Yes[a]	Yes	No	Yes
Existentials	No	Yes	No	Yes
Nested quantification	Yes	Yes	No	Yes
Displays partial counterexamples	N/A	N/A	No	Yes
Haskell 98/2010	Partial[b]	Compatible	Compatible	No[c]

[a] Functional value is wrapped in a modifier at its quantification binding if showing or shrinking is required.
[b] Originally Haskell 98 compatible but functional values modifier requires GADTs.
[c] Requires Haskell extensions: GADTs, type families and flexible contexts.

3.5.1 Runtime performance

We have compared this implementation's runtime performance with the previously published Lazy SmallCheck using the benchmark properties from Runciman et al. (2008). Experiments performed using GHC 7.6.1 with `-O2` optimisation on a 2GHz quad-core PC with 16GB of RAM show very little difference in execution times between the two encodings.

We had expected to see a performance improvement due to the removal of the deserialization stage for generating terms. While there are minor improvements in some examples, it seems that GHC was particularly adept at optimising the deserialization process.

3.5.2 Alternative strategies

Lazy SmallCheck explores depth-bounded spaces of test-data values. An alternative approach would be to explore *size-bounded spaces*, such as those faciliated by Feat (Duregård et al., 2012). The benefit is that the search space grows more slowly for each increase in maximum size than it does for maximum depth. However, the technique used by Duregård et al. (2012) to efficiently generate these spaces is not especially compatible with the demand driven search of Lazy SmallCheck.

3.5.3 Functional values

The original QuickCheck paper (Claessen and Hughes, 2000) explains how functional test values can be generated through the *Arbitrary* instance of functions with a *Coarbitrary* instance of argument types. At this stage, QuickCheck could not display the failing example without bespoke use of the *whenFail* property combinator. QuickCheck has since gained the ability not only to display functional counterexamples but also to reduce their complexity through *shrinking*. Claessen (2012) achieves this by transforming functions generated using the existing *Coarbitrary* technique into tries.

Claessen's formulation of tries slightly differs from ours. Existential types are used in place of type families and there is no provision for non-strict functions. Partiality of functions is explicitly expressed instead of being a result of partially defined tries. Claessen also requires that functions are wrapped in a *'modifier'* at quantification binding. This *Fun* modifier retains information for showing and shrinking at the expense of a slightly more complex interface presented to users.

In Lazy SmallCheck, on the other hand, we directly generate a trie and then convert it into a Haskell function. A pretty-printed representation of the trie is stored at the time of generation and retrieved for counterexample output. The SmallCheck representation of functional values uses a *coseries* approach, analogous to QuickCheck's *Coarbitrary*. However, functional values are displayed by systematically enumerating arguments.

3.5.4 Existential and nested quantification

As previously discussed in Section 3.1, it is not appropriate to use QuickCheck for existential quantification as the chance of randomly selecting an (often) single witness is negligible. The previous design of Lazy SmallCheck made it difficult to conceive of a refutation algorithm that could handle the nested quantification required to make existential properties useful. The use of the *Partial* values functor in this implementation gives statically typed guarantees that term refinements are performed at the correct quantifier nesting.

3.5.5 Benefits of laziness

Runciman et al. (2008) discussed the benefits and fragility of exploiting the laziness of the host language to prune the test-data search space. When applied to functional values, we see further benefits. The partiality of a trie representation corresponds directly with the partiality of the function it represents. Whereas Claessen (2012) needs to shrink total function to partial functions, the latest Lazy SmallCheck has partial functions as a natural result of its construction.

3.6 SUMMARY

This chapter has described the extension of Lazy SmallCheck with several new features: *(1)* quantification over functional values, *(2)* existential and nested quantification in properties and *(3)* the display of partial counterexamples.

Properties that quantify over functional values occur often in higher-order functional programming. Similarly, many properties may involve existential quantification and even nesting of quantification within property definitions. The examples in this chapter have demonstrated the power of a tool that can find counterexamples for such properties.

This chapter takes an *extensional* view of functional values, characterising them as mappings from input to output. An alternative would be to characterise functions *intensionally* as lambda abstractions or other defining expressions, perhaps allowing recursion (Katayama, 2007; Koopman and Plasmeijer, 2010). We would expect the generic machinery for typed functional series to be more complex. Also, when functions are needed as test values, alternative definitions of the same extensional function are not particularly useful. This will be resolved in the next chapter.

4 PROGRAMS AS TEST-DATA

We describe experiments generating functional programs in a core, simply-typed, first-order language with algebraic data types. Candidate programs are generated freely over a syntactic representation with positional names. Static conditions for program validity and canonical representatives of large equivalence classes are defined separately.

4.1 INTRODUCTION

In this chapter, we use Lazy SmallCheck to enumerate all small test programs that are *valid* (well-formed, well-scoped and well-typed, Section 4.2.3), *canonical* (of a regular form detailed in Section 4.3) and *terminating* (also Section 4.3).

Rather than directly constructing programs that satisfy these conditions, we freely generate abstract syntax trees and filter out those for which some required condition does not hold. With careful representation choices for the freely generated abstract syntax, a lazy and condition-driven approach to test generation can efficiently and effectively prune large classes of unwanted test programs. Ideally, we would treat these canonical programs as representatives of equivalence classes from which test programs shall be selected. However, in this chapter, we shall focus on exploring these equivalence classes through the canonical programs.

4.2 GENERATING VALID PROGRAMS

4.2.1 Our core language

We choose to work with a simply-typed, first-order core functional language with algebraic data types. The following example is a program in this language. The function `add` is addition over `D`, a representation of the Peano numerals. The expression to be evaluated, indicated by `>`, adds one and one.

```
data D = Zero | Succ D
add m n = case m of
          Zero   -> n
          Succ p -> Succ (add p n)
> add (Succ Zero) (Succ Zero)
```

```
data Pro    = Pro (Seq1 Nat) Exp (Seq RedDef)
data RedDef = Lam Nat Bod
data Bod    = Solo Exp
            | Case Exp (Seq1 Alt)
data Exp    = Var VarId
            | App DecId (Seq Exp)
data VarId  = Arg Nat
            | Pat Nat
data DecId  = Con Nat
            | Red Nat
data Alt    = Nat :⟶: Exp
```

Figure 4.1: Initial definition of our core language.

In order to generate programs in this language, we first define a data type for its abstract syntax, as in Figure 4.1. A program *Pro cds e rds* consists of a single data type definition represented as a sequence *cds* of one or more constructor arities, a main expression *e* to be evaluated and zero or more top-level value definitions *rds* whose applications are reducible. A top-level value definition *Lam ar b* is a lambda abstraction of arity *ar*. The body may be just a single applicative expression *Solo e* or it may be a case expression *Case e as* with alternatives for different constructions of the subject *e*.

Expressions are, as usual, recursively composed applications with either variables or zero-arity applications as leaves. Variable references are explicitly tagged: *Arg* for argument variables and *Pat* for pattern variables in alternatives. Applied references are also tagged: *Con* for constructors and *Red* for top-level names whose applications are reducible. These are all referenced by the natural-number positions of their definitions.

4.2.2 Free generation

Using SmallCheck (Runciman et al., 2008) we can now define functions to enumerate all values of these AST data types bounded by a given depth of construction. The *Serial* instances are defined in Figure 4.2.

The type *Seq a* is synonymous with the list type $[a]$ but the depth-bound is the same for all elements of the list — a *Seq a* list bounded by depth d has at most d items, each of which has depth at most $d - 1$. The *Seq1* variant is for lists with at least one element.

It is often convenient not to count simple tags or tupling structures when determining the depth of a construction. The compositions with *depth* 0 are for that purpose.

```
instance Serial Pro where
  series = cons3 Pro ∘ depth 0
instance Serial RedDef where
  series = cons2 Lam ∘ depth 0
instance Serial Bod where
  series = (cons1 Solo ∪ cons2 Case) ∘ depth 0
instance Serial Exp where
  series = cons1 Var ∪ cons2 App
instance Serial VarId where
  series = (cons1 Arg ∪ cons1 Pat) ∘ depth 0
instance Serial DecId where
  series = (cons1 Con ∪ cons1 Red) ∘ depth 0
instance Serial Alt where
  series = cons2 (:⟶:) ∘ depth 0
```

Figure 4.2: The series generators for our initial syntactic representation.

Let's run the *Pro* series generator with increasing depth bounds, and count the number of programs generated.

```
 >>> [ length (series i :: [Pro]) | i <- [0..] ]
[0, 4, 3504, 27700575980220, ...
```

What do the four *Pro* values at depth one look like? These can be rendered as follows, with the convention that arguments are renamed `x`,`y`,..., pattern variables `p`,`q`,..., constructors `A`,`B`,... and top-level functions `f`,`g`,....

```
data D = A      data D = A      data D = A      data D = A
> x             > p             > A             > f
```

As these programs are freely generated from the abstract syntax type, they are as yet unconstrained by any static semantics. Only one of them is valid — the third one. At depth two there are already thousands of similar-looking *Pro* values, hardly any of which are valid. Beyond depth two our machines are overwhelmed by the task of enumeration.

```
  -- Test a program is well-scoped and arity consistent.
valid :: Pro → Bool
valid (Pro (Seq1 cons) m (Seq eqns)) = valide 0 0 m ∧
                                       all validr eqns
  where
    -- Test a reducable is well-scoped and arity consistent.
    validr (Lam a (Solo e))           = valide a 0 e
    validr (Lam a (Case s (Seq1 alts))) = valide a 0 s ∧
      and [indexThen c cons (λp → valide a p e)
           | (c :⟶: e) ← alts]
    -- Test an expression is well-scoped and arity consistent.
    valide a _ (Var (Arg v))     = a ≢ 0 ∧ v < a
    valide _ p (Var (Pat v))     = p ≢ 0 ∧ v < p
    valide a p (App d (Seq es)) = valida d (N $ length es) ∧
                                  all (valide a p) es
    -- Test an application is well-scoped and arity correct.
    valida (Con c) n = indexThen c cons (λn'          → n ≡ n')
    valida (Red f) n = indexThen f eqns (λ(Lam n' _) → n ≡ n')
  -- Index an element from a list and apply predicate. Default to False.
indexThen :: Nat → [a] → (a → Bool) → Bool
indexThen (N i) xs f = (¬ ∘ null) xs' ∧ head xs'
  where xs' = map f (drop i xs)
```

Figure 4.3: Validity of positional programs.

4.2.3 Validity test

Only one of the programs generated at depth one was valid. The other three referred to *undefined* variables or functions. At greater depths another form of invalidity can occur: there may be *arity disagreement* between uses and definitions. We must avoid, or cut short, the work of generating such invalid programs.

We can define a predicate *valid*, as in Figure 4.3. The auxiliary functions *validr*, *valide* and *valida* test the validity of reducibles, expressions and applications respectively. The first two arguments to *valide* are the enclosing scopes for argument and pattern variables.

If we test the property $\lambda p \rightarrow valid\ p \implies True$, of the 3,504 syntactically generated programs at depth two, just 160 are found to be valid. So even this simple validity check greatly reduces the number of programs to be tested. But as things stand, we still have to generate a large number of invalid programs, only to reject them as test cases.

4.2.4 Lazy free generation

The problem of generating test cases that satisfy conditions was a large part of the motivation for *Lazy* SmallCheck (Runciman et al., 2008). This tool applies conditions to *partially defined* values. If a test value is sufficiently defined to allow a condition to be evaluated to True (or to False), then it is known from this single evaluation that *all possible refinements* of this test value will also satisfy (or fail to satisfy) the condition. If the partiality of a test value makes the condition undefined, the test value is refined at exactly the place needed for evaluation of the condition to proceed further.

In principle, Lazy SmallCheck might run *more* test cases than those seen under SmallCheck for the same condition — since it tests partial values as well as total ones. But in practice, where there is a structural condition that most tests do not satisfy, Lazy SmallCheck uses many fewer tests.

If we again test the property $\lambda p \rightarrow \mathit{valid}\ p \implies \mathit{True}$, but this time using Lazy SmallCheck, just 187 tests are needed to obtain the same 160 programs at depth two. That is just under 5% of the tests required under SmallCheck.

4.3 CANONICITY

If we could only test a compiler using just two source programs, it would be a better test if the two programs really were quite distinct, not just insignificant variations of each other. The same argument applies even if we can use a large number of test programs. Resources are always limited. So we don't want to waste them by testing umpteen versions of essentially the same program.

We shall use several principles to define *canonical* programs. Each of these programs is a unique representative of a whole class of essentially equivalent programs. The principles of canonicity are discovered through the analysis of programs being generated.

For the purposes of testing, the set of canonical programs would ideally only be used as identifiers of their equivalence classes. It is from these equivalence classes that test candidates should be selected, preferably at random. This will be discussed further in section 9.3. However, in this chapter we shall focus on producing the set of canonical programs.

4.3.1 A note about parallel conjunction

Lazy SmallCheck (Runciman et al., 2008) exposes a *parallel conjunction* operator, $(\wedge_{par})$, which is falsified if *either* of its conjuncts are false. In contrast, standard conjunctions $(\wedge)$ returns $\perp$ if its first conjunct is $\perp$ even if its second is *False*. Properties defined using parallel conjunction instead of standard conjunction

can reduce the amount of structure that must be expanded to reach a result and therefore decrease the number of tests performed.

In the definitions of our canonicity principles, we often use parallel conjunction *between* each principle and any component of a principle that will independently traverse the abstract syntax tree. For instance, in Figure 4.4 we use parallel conjunction between the tests of ordering over constructor arities, equations and case alternatives.

4.3.2 Principles of ordering and complete reference

The two programs below perform the same computation under the obvious isomorphism between their data types. The only difference between them is the *ordering* of constructors, function definitions and case alternatives.

```
data D  =  A | B D D
f x y   =  case x of
  A     -> y
  B p q -> B p (g q y)
g x     =  case x of
  B p q -> p
> g (f A (B A A))
```

```
data D  =  A D D | B
f x     =  case x of
  A p q -> p
g x y   =  case x of
  A p q -> A p (g q y)
  B     -> y
> f (g B (A B B))
```

A canonical representative of both programs respects an ordering for each of these things. Assuming the standard, automatically derived instances of *Ord* for our AST data types, a canonical ordering predicate for programs is given in Figure 4.4.

The orderings over equations and alternatives are irreflexive; we forbid duplicate definitions. The ordering over constructor arities is not; we permit more than one constructor of the same arity.

The following programs are also in direct correspondence. There is a duality so far as the roles of the constructors `A` and `B` are concerned, and the arguments of function `f` are flipped.

```
data D = A | B
f x y = case x of
        A -> A
        B -> y
> f B B
```

```
data D = A | B
f x y = case y of
        A -> x
        B -> B
> f A A
```

So here is a further ordering requirement in canonical programs. Constructors of equal arity must be *first used* in the program in the same order as they are

```
canonicalOrder (Pro (Seq1 cons) _ (Seq eqns)) =
    -- Non-strict ordering of constructor arities
  orderedBy (⩽) cons ∧par
    -- Strict lexicographic ordering of equations
  orderedBy (<) eqns ∧par
    -- Strict ordering of case-alternatives by constructor
  and [ orderedBy (<) [c | (c :⟶: _) ← alts]
      | (Lam _ (Case _ (Seq1 alts))) ← eqns]
orderedBy :: (a → a → Bool) → [a] → Bool
orderedBy f (x : y : zs) = f x y ∧ orderedBy f (y : zs)
orderedBy _ _            = True
```

Figure 4.4: Predicate for the ordering of constructors, equations and alternatives.

declared in the data type. And function arguments must be *first used* in the function body in the order given by their argument positions.

Further, for any program that declares *unused* constructors, arguments or pattern variables there is a simpler equivalent program without them. In a canonical program, all constructors and arguments are used.

Finally, a program with *unused function definitions* also has a smaller equivalent without them. In a canonical program, all functions can be reached by a static call-chain from the main expression. See Section 4.3.8 for further discussion of dead code.

After we impose all these ordering and complete-reference conditions, we have just two programs at depth two, generated by Lazy SmallCheck as a result of 109 tests. And at depth 3, instead of an overwhelming number of programs, just 4,413 programs are produced as a result of 24,373,980 tests.

4.3.3 Unorderable equations

Consider the following programs that *do not* satisfy the equation-ordering condition.

```
data D = A | B A
f x = B (g x)
g x = B (f x)
> f A
```

```
data D = A | B A
f x = B (g x)
g x = B (f x)
> g A
```

In the current positionally-referenced representation, these programs have *no canonical form*. Reversing the equation ordering simply gives the other

program. Our solution for now is to *limit the number of top-level definitions to two* and change the referencing scheme as follows. Within a top-level definition reference is either recursive or else it references the other top-level definition: *Self* and *Other*. Within the main expression, we keep positional naming. i.e. 0 and 1.

As both of these reference models can be implemented with Boolean values, the *Red* constructor is changed to hold *Bool* instead of *Nat*. The definition of *valid* also needs to be changed to account for the new referencing scheme. The ordering predicates work without modification.

4.3.4 Principle of depth balance

To reach a rich space of small test programs, we need to generate function bodies at around depth four or five. But we do *not* need data types with four or five constructors, each with four or five arguments! Nor do we need multiple high-arity function declarations.

At depth n, the default syntactic generators give between one and n constructors. The constructors and functions each have $arity \leq n$. Not only is this signature space far richer than we need to express interesting programs — LISP has taught us that — but also the depth limits largely prevent *uses* of these declarations from being generated anyway.

Therefore, mirroring the top-level *two function limit*, the number of constructor declarations and the arities of declarations are capped at two. This could be implemented using a further condition but another approach will be outlined in Section 4.3.6.

4.3.5 Principle of caller/callee demarcation

Wherever there is an application of a defined function, there may be different ways to split work between caller and callee. A canonical program should make this split only in standardised ways.

Both caller and callee should do *something*. The caller must do something: it cannot just be the application of the callee to some of the caller's arguments, or else any application of the caller could more simply be an application of the callee. The principle of complete reference excludes many cases, but we also exclude as a body any application of a function to exactly the same arguments. The callee must also do something: a function body cannot simply be one of the arguments, or else any application could be replaced by a subexpression. Again the principle of complete reference already excludes most cases, but we also exclude the identity.

Even in our original program representation, we had a form of caller/callee constraint: case expressions can only occur outermost in a function body. So

```
data ProR    = ProR ExpR (Seq0'2 BodR)
data BodR    = SoloR ExpR | CaseR (AltR, AltR)
data ExpR    = VarR VarIdR | AppR DecIdR (Seq0'2 ExpR)
data VarIdR  = ArgR Bool   | PatR Bool
data DecIdR  = ConR Bool   | RedR Bool
data AltR    = NoAltR      | AltR ExpR
```

Figure 4.5: Nonredundant representation of our core language.

the callee does the case distinction. In canonical programs, the caller computes the case subject: that is, a case subject is just an argument variable, and by the ordering principle, it must be the first argument.

This too could be implemented by a further condition, but we use another approach, as the following section explains.

4.3.6 Principle of nonredundant representation

It is pleasing that Lazy SmallCheck can prune away the 3,502 invalid or non-canonical programs of depth at most two by running only 109 tests, finally delivering for us the two interesting test programs — or more precisely, two equivalence classes of programs. But the very high proportion of *Pro* values that fail the conditions does prompt a question: would a further change of representation enable us to generate fewer invalid or non-canonical programs in the first place?

We have already established that canonical case subjects are first arguments. So in our new representation the case subject can be omitted.

For a program to be valid, all uses of constructors or functions must match declared data type and function arities. In a canonical program with complete reference, it follows that the data type can be determined from the other parts of the program, and the arity of each function can be determined from its body. So instead of generating a data type definition and function arities, and testing for valid and complete uses, we need only generate a main expression and function bodies.

The cap of two on the number of constructors and functions can also be encoded in the sequence representation types in programs, and in *Bool* index types for declarations. With function arities bounded by two a *Bool* index also suffices for argument variables. Figure 4.5 details the new representation.

Case-alternative patterns now reference constructors according to their position, doing away with the need for a separate ordering condition for alternatives.

The arity of functions can be deduced by finding the maximum argument in the function body. The data type definition can be inferred by combining information about program constructor applications and the maximum pattern variable in constructor alternatives. Conditions are still used to prune away non-canonical programs that are not precluded by the nonredundant representation.

The change to a nonredundant representation dramatically reduces the number of tests required at each depth. At depth 3 (analogous to the previous representation's depth 2), only 25,393 tests are required to reduce a space of 2,371,256 programs to 11 canonical representatives. At depth 4, analogous to the previously unattainable depth 5, it takes 28,311,473 tests to find 423,582 canonical programs.

4.3.7 Principle of live computation

Most interesting functional programs are recursive. But some recursively defined functions can unproductively fail to terminate. For example, here are two programs generated at depth 4.

```
data D  =  A                    data D  =  A | B D
f       =  g A                  f       =  B f
g x     =  case x of            g x     =  case x of
             A -> f                           A   -> x
> f                                          B p -> g p
                                > g f
```

To exclude programs such as the one on the left, we add the condition that any recursive applications are either beneath a constructor, or else descend into the construction of a recursive argument. At depth 3, this simple termination condition does not reduce the number of programs produced but it does reduce the number of tests required to 19,099. At depth 4, only 74,414 canonical programs are now produced after 20,550,413 tests.

This still leaves some non-terminating programs such as the one on the right. (View `D` as Peano numerals, `f` as infinity and `g` as a semi-test for finite numbers.) A far more sophisticated condition (e.g. Abel, 2000) would be needed to eliminate such programs yet allow useful recursion.

For now, we have decided to accept that some unproductive programs will remain. A more sophisticated condition would require significant extra machinery and adversely affect lazy pruning performance. However, property testing must allow for the possibility of an unproductive program.

4.3.8 Principle of live code

The following programs are among those generated at depth 3. They are indistinguishable in their execution as the `B` case alternatives are never used. Some form of data-flow analysis is needed to detect dead code.

```
data D = A | B
f x y
  = case x of
      A -> y
> f A B
```

```
data D = A | B
f x y
  = case x of
      A -> y
      B -> x
> f A B
```

```
data D = A | B
f x y
  = case x of
      A -> y
      B -> A
> f A B
```

Dynamic evaluation of candidate test programs, followed by a simple reachability analysis, detects dead code more accurately than reachability analysis alone. We must avoid unbounded computation arising from recursive applications, but to avoid unfolding all recursive calls would limit results too much. Our solution is *single-shot recursion*: on any call path we evaluate at most two applications of the same function.

The bounded evaluation traverses the abstract syntax tree in *normal order*, contrasting with the other *in-order* conditions. Validity checks can therefore be bypassed due to the use of Lazy SmallCheck's *parallel conjunction* operator. As validity is required for evaluation, a partial validity checker is integrated into the dead code checker.

Although the live-code condition supersedes the function-reachability and constructor-use of Section 4.3.2, it is still worth applying all these conditions. The combination of different traversal orders may prune failures sooner.

Eliminating programs with dead code results in another dramatic fall in tests; depth 3 requiring only 2,731 tests and depth 4 only 445,791 tests. Now just four canonical programs remain at depth 3. These are the constant `A` program and the following:

```
data D = A | B
f x = case x of
      A -> B
> f A
```

```
data D = A
f x y = case x of
        A -> y
> f A A
```

```
data D = A | B
f x y = case x of
        A -> y
> f A B
```

The leftmost program could be interpreted as partial inversion with `D` as the Boolean type. Both other programs are partial conjunction, where `A` is True and `B` is False, with different inputs. Alternatively, these could be viewed as partial disjunction where `A` is False and `B` is True.

At depth 4, we have just 64 programs that satisfy all these principles of canonicity and validity.

Table 4.1: Performance of non-redundant representation at depth 3.

Conditions	Execution time	Tests required	Programs
Validity	2643ms	138,617	855
+ Ordering + Use	690ms	34,745	124
+ Caller/Callee	580ms	25,393	11
+ Live Computation	437ms	19,099	11
+ Live Code	72ms	2,731	4

4.4 PERFORMANCE

So far, we have discussed performance abstractly, with regard to the number of tests to reach a set of desirable programs. In this section, we shall also consider execution time. All figures were obtained using GHC 7.0.3 on 2GHz dual-core PC with 4GB of RAM.

Table 4.1 shows performance figures when applying the various conditions at depth 3 of the non-redundant representation. The initial freely generated space contains 2,043,136 'programs'. Execution times are measured using the *Criterion* (O'Sullivan, 2011) benchmarking library, averaging 100 measurements and ensuring a 0.95 confidence interval. As each additional condition is applied, the number of tests required to reach a set of desirable programs falls. This trend is mostly mirrored by a fall in execution time. However, execution time does not fall quite as rapidly as the number of tests performed. The time per test lengthens as the number of conditions increases. In fact, the *mean execution time per test* increases by 38% from validity to the full suite of conditions for canonicity.

Enumerating all canonical programs at depth 4 takes approximately 15 seconds. At depth 5, it takes around 3 hours to produce the 310,003 canonical programs.

4.5 APPLICATIONS

We use these canonical programs to investigate the correctness properties of language implementations and program optimisations. The first example produces a small program that exposes the differences between static binding and dynamic binding. The second investigates some correctness properties of compiler optimisations both in terms of semantic preservation and performance improvement.

```
prop_bind :: Pro → Bool
prop_bind e = isJust static ∧ isJust dynamic  ⟹  static ≡ dynamic
  where static  = evalFor 1000 False e ≫= return ∘ forceResult 5
        dynamic = evalFor 1000 True e ≫= return ∘ forceResult 5
```

Figure 4.6: A mistaken equivalence between static and dynamic binding.

4.5.1 Static vs. dynamic binding

Suppose we implement different semantics for our source language. One version uses *static binding*, evaluating arguments in the environment of the application call. The other uses *dynamic binding* where arguments are evaluated in the environment of the argument reference.

The generated programs are evaluated under each semantics up to a given maximum derivation-tree depth and the results are compared under equality. This property is defined as *prop_bind* in Figure 4.6. Testing discovers a small example program at depth 4, for which static binding and dynamic binding produce different results.

```
data D = A | B D
f = g A A
g x y = case x of
        A   -> B y
        B p -> g p x
> g f f
```

Under static binding, the program returns `B (B A)` as we would usually expect. However, under the dynamic binding semantics, the program returns `B A`. In the recursive call to `g`, the environment contains $\{x \mapsto p, y \mapsto x, p \mapsto A\}$ when variable `y` is referenced.

4.5.2 Optimisations on a Sestoft Abstract Machine

Sestoft (1997) details the derivation of a series of abstract machines of increasing efficiency. These abstract machines evaluate expressions written in a core higher-order functional language. A simple transformation converts our core first-order language into a form that can be executed by the Sestoft Mark 2 abstract machine.

Our goal is to verify a simple program transformation that non-recursively inlines function applications. In this case, we wish to ensure not only *semantic*

$$
\begin{aligned}
&\mathit{prop\_inline\_sem} :: \mathit{ProR} \to \mathit{Bool} \\
&\mathit{prop\_inline\_sem}\ p = \mathit{isJust}\ (\mathit{haltState}\ r0) \implies \\
&\qquad\qquad\qquad \mathit{haltState}\ r0 \equiv \mathit{haltState}\ r1 \\
&\quad \textbf{where}\ r0 = (\mathit{traceFor}\ 1000 \circ \mathit{translate})\ p \\
&\qquad\quad r1 = (\mathit{traceFor}\ 1000 \circ \mathit{translate} \circ \mathit{opt\_inline})\ p \\
&\mathit{prop\_inline\_opt} :: \mathit{ProR} \to \mathit{Bool} \\
&\mathit{prop\_inline\_opt}\ p = \mathit{isJust}\ (\mathit{haltState}\ r0) \implies \\
&\qquad\qquad\qquad \mathit{length}\ r0 \geqslant \mathit{length}\ r1 \\
&\quad \textbf{where}\ r0 = (\mathit{traceFor}\ 1000 \circ \mathit{translate})\ p \\
&\qquad\quad r1 = (\mathit{traceFor}\ 1000 \circ \mathit{translate} \circ \mathit{opt\_inline})\ p \\
&\mathit{translate} :: \mathit{ProR} \to \mathit{SestExpr} \\
&\mathit{traceFor} :: \mathit{Int} \to \mathit{SestExpr} \to [\mathit{SestState}] \\
&\mathit{haltState} :: [\mathit{SestState}] \to \mathit{Maybe}\ \mathit{SestExpr}
\end{aligned}
$$

Figure 4.7: Predicates for testing inlining transformation.

equivalence but also *optimisation of reduction steps*. These are formally defined as *prop_inline_sem* and *prop_inline_opt* respectively in Figure 4.7.

At depth 5, the semantic equivalence property is satisfied by all 310,003 canonical programs. However, the following counterexample is found for the optimisation property. If no inlining is performed then this program takes 44 steps to reduce to normal form. But if inlining is applied it takes 46 steps.

```
f x   = case x of
          A   -> B x
          B p -> g x p
g x y = case x of
          A -> f y
          B p -> x
> f (g A A)
```

The reason is as follows. In the original, `g A A` is only evaluated once but after inlining it is evaluated twice. The shared evaluation of `x` in the body of `f` is lost.

4.6 RELATED WORK

A survey from the late 1990s (Boujarwah and Saleh, 1997) discusses and classifies a range of techniques for generating test programs. The papers cited generally use advanced generating grammars to ensure that only "semantically correct" (valid) programs are produced. A few authors generate test programs freely over context-free or EBNF grammars but with the stated aim of testing a compiler's syntax checker.

As discussed in Section 2.4, Palka et al. (2011) describes the use of QuickCheck to generate random lambda terms for compiler testing. De Bruijn (1972) indexing is used to avoid problems of equivalence up to renaming. Aside from the use of random lambda terms, as opposed to exhaustively enumerated small equational programs, another significant difference from the approach reported here is that Palka et al. rely on a generating context including the signatures of pre-defined functions.

Other functional-programming researchers have looked into program enumeration. For example, Katayama (2007) enumerates typed lambda terms. The motivation is to provide exhaustive search for appropriately typed expressions during program synthesis. Katayama highlights the advantages of a de Bruijn representation, and the importance of excluding "equivalent expressions which cause redundancy in the search space and multiple counting". In this work too, the generator generates terms applying a library of pre-defined functions, and one of the equivalance-avoiding techniques is to apply known simplification laws for these functions. But the discussion notes a need to do more to eliminate duplicate or equivalent solutions.

4.7 SUMMARY

Our aim has been to enumerate valid and canonical programs for the purposes of compiler verification. We have shown that large spaces of freely generated terms can be pruned effectively to yield 'interesting' programs. Exploration of the search space indicates that Boolean programs such as partial inversion, conjunction and disjunction appear at depth 3. Canonical programs involving Peano numerals (e.g. addition) and lists (e.g. append) emerge at depth 6.

It is unclear at what depth the generated programs cease being *'interesting'* for the purposes of identifying equivalence classes. Certainly, even with these canonicity principles pruning the search space, the additional depth quickly becomes unmanageable. It may be better to use a size-bounded approach like Feat (Duregård et al., 2012).

This chapter roughly mirrors the process by which the principles were discovered. First, an algebraic data type for the abstract syntax is defined and

a free generator is created using (Lazy) SmallCheck combinators. Through the observation of the resulting programs, conditions are defined to eliminate invalid and non-canonical programs. The representation is reconsidered to eliminate the redundancy that allows the invalid and non-canonical terms to arise. And so the procedure repeats. Implementation details are occasionally reevaluated to account for the interactions of the different conditions.

We have discovered several principles of canonicity for our first-order language and dramatically reduced the problem size. We expect that further investigation of the currently generated programs will reveal new principles of canonicity or more restrictive variations of existing conditions.

We applied our testing technique to investigate several properties relating to evaluation, compilation and optimisation. The results obtained are encouraging. However, more complex applications motivated our work: we wish to investigate the correctness and improvement properties of supercompilers. This is the focus of the upcoming chapters.

Part III

VERIFICATION OF A SUPERCOMPILER

5 A REVIEW OF SUPERCOMPILATION

This chapter introduces the concept of supercompilation. Literature on the topic is reviewed through the construction of a literate supercompiler implementation. This exercise prompts discussion of design choices and verification concerns. This reference implementation will be used to design the verified supercompiler.

5.1 INTRODUCTION

Supercompilation (Turchin, 1979) is a metaprogramming technique that, at compile-time, reduces programs until an unknown value is required or some other termination condition is met. The algorithm then proceeds by case analysis, reusing previously calculated results when appropriate. The output of a supercompiler is a *residual program*.

For example, consider the following function definition. It is written in a small core functional language described in detail in Section 5.2.

```
letrec map = \f xs ->
   case xs of { Nil       -> Nil;
                Cons y ys -> Cons (f y) (map f ys) }
in \f g xs -> map f (map g xs)
```

This function composes two *map* operations over a list, *xs*. Unfortunately, the inner map deconstructs the input and constructs the intermediate list only for the outer map to deconstruct that intermediate list and construct the output list. Applying the supercompiler described in this chapter yields the more efficient definition:

```
letrec h0 = \f g xs ->
  case xs of { Nil       -> Nil;
               Cons y ys -> Cons (f (g y)) (h0 f g ys) }
in h0
```

The composed *map* operations have been *fused* into one function, eliminating the intermediate list. In a program that applies such higher-order functions, they may be specialised for given functional arguments with further corresponding performance gains at execution time.

Supercompilation is distinct from *partial evaluation* (Bjorner et al., 1988; Ershov, 1982; Jones, 1988; Jones et al., 1993) as it does not require input data to

$$
\begin{aligned}
\langle exp \rangle ::= & \langle varId \rangle \mid \langle conId \rangle \\
& \mid \lambda \langle varId \rangle^{+} \rightarrow \langle exp \rangle \\
& \mid \langle exp \rangle \ \langle exp \rangle \\
& \mid \textbf{let} \ \langle varId \rangle = \langle exp \rangle \ \textbf{in} \ \langle exp \rangle \\
& \mid \textbf{letrec} \ \langle varId \rangle = \langle exp \rangle \ \textbf{in} \ \langle exp \rangle \\
& \mid \textbf{case} \ \langle exp \rangle \ \textbf{of} \ \{ \langle alt \rangle^{+} \} \\
& \mid (\langle exp \rangle) \\
\langle alt \rangle ::= & \langle conId \rangle \ \langle varId \rangle^{*} \rightarrow \langle exp \rangle
\end{aligned}
$$

Figure 5.1: Concrete syntax for core language.

be effective. While the resulting programs are similar to those that have had *deforestation* (Wadler, 1990) and Mitchell and Runciman (2009) -style *defunctionalisation* applied, the process by which this is achieved is quite different.

The following sections describe an implementation of a supercompiler for a small core language. Design choices are contrasted with other implementations in the literature.

5.2 THE CORE LANGUAGE

The core source language for the supercompiler developed in this chapter is an extended lambda calculus. It includes recursive and non-recursive local definitions and case discrimination over algebraic data structures. The concrete syntax for this language can be found in Figure 5.1. An abstract syntax is defined in the Haskell language in Figure 5.2. Note that:

(a) Every node in the abstract syntax tree is *tagged* with a value of parameter type t. As we shall see in Sections 5.3.1 and 5.3.2, this tagging allows the operational semantics code to be reused in the supercompiler.

(b) While the concrete syntax uses *named variables*, the abstract syntax uses a *locally nameless* (Charguéraud, 2012; McBride and McKinna, 2004) representation. Bound variables are referenced by *de Bruijn indicies* (de Bruijn, 1972) and free variables are referenced by their heap locations. This representation simplifies abstract syntax tree manipulations. Type signatures for some of these manipulation functions can be found in Figure 5.14 on page page 75 at the end of this chapter.

(c) Applications can only be made to variables. This forces application argument expressions onto the heap, ensuring results of their evaluation can be shared.

```
  -- Tagged core language expressions
data Exp t  = (Exp' t)^t
data Exp' t = Var Ref
            | Con Id [(t, Ref)]
            | \•→ (Scope t)
            | Exp t ⊔ Ref
            | let • = Exp t in Scope t
            | letrec • = Scope t in Scope t
            | case (Exp t) of (Alts t)
  -- Case alternatives
type Alts t = [(Id, Scopes t)]
  -- Variable references
data Ref = Fre HP | Bnd Ix
  -- de Bruijn indicies
newtype Ix = Ix Int
  -- Binding scopes
newtype Scope  t = Scope (Exp t)
data    Scopes t = Open (Scopes t) | Closed (Exp t)
type Id = String
```

Figure 5.2: Tagged abstract syntax data type for core language.

```
  -- Abstract machine states
data State t = ⟨ (Heap t) | Exp t | Stack t ⟩
  -- Heap and heap pointers
type Heap t = (HP, Map HP (IsRec, Exp t))
newtype HP = HP Int
  -- Recursive definition flag
data IsRec = NonRec | Rec
  -- Control stack
type Stack t   = [(t, StkElem t)]
data StkElem t = APP HP | UPD HP | CAS (Alts t)
  -- Successful values are closures.
type Value t = Maybe (Heap t, Exp t)
```

Figure 5.3: Data types used by the operational semantics.

A *call-by-need* operational semantics for this language is presented in Figure 5.4, utilising the data types and auxiliaries in Figures 5.3 and 5.14.

The *State t* data type in Figure 5.3 represents the abstract machine state in our operational semantics. The states are parameterised by a tag type t to match our tagged expression trees. A state consists of a heap, a control (or focus) expression and a control stack.

The heap keeps track of the next fresh heap pointer and contains a mapping from heap pointers to expressions. These expressions use a flag to recall if they were introduced through a recursive let-binding or a non-recursive let-binding. The control stack is a tagged list of stack elements, defined by the *StkElem t* data type. It records expression contexts such as applications, heap updates and case distinctions.

The semantics are presented as a Haskell function, *step*, so that *(1)* the interpreter, *run* also defined in Figure 5.4, can be easily defined and *(2)* it can be easily reused as part of the supercompiler. The semantics are based on the Sestoft (1997) mark 3 abstract machine. An abstract machine state consists of a heap, an expression under focus and a control stack. To evaluate an expression, an initial machine state, with an empty heap and stack, can be constructed using *initialState*. The $\cdot\ [\star \mapsto \cdot]$ function is used to place expressions onto the heap, returning the a pointer to the expression on the heap, *nxt*, and the updated heap.

5.3 A SUPERCOMPILER FOR THE CORE LANGUAGE

A supercompiler can be decomposed into four components: *(1)* a *normaliser* that simplifies terms, *(2)* a *terminator* that prevents non-terminating simplification, *(3)* a *splitter* that produces smaller terms for further supercompilation and *(4)* a *memoiser* that reuses the results of supercompiling equivalent terms. These are integrated through an overall *controller* algorithm. Others authors may use different terms but the terminology use here is similar to that used by Bolingbroke and Peyton Jones (2010).

5.3.1 Normalisation

Many past formulations of supercompilation (Jonsson and Nordlander, 2008; Klyuchnikov, 2010; Mitchell, 2010; Mitchell and Runciman, 2008; Reich et al., 2010; Sørensen et al., 1996) use custom simplification rules to normalise terms. More recently, there has been a trend towards *normalisation-by-evaluation* (Bolingbroke and Peyton Jones, 2010). In this approach the term is evaluated under an operational semantics, but restricting the application of rules that may involve

```
  -- Step function for operational semantics
step :: State t → Maybe (State t)
step ⟨ Γ | (Var (Fre p))^t | σ ⟩                    -- (Rule: var_0)
  = do (_, x) ← lookup p Γ
       Just ⟨ Γ | x | (t, UPD p) : σ ⟩
step ⟨ Γ | (x ⊔ (Fre p))^t | σ ⟩                    -- (Rule: app_0)
  = do Just ⟨ Γ | x | (t, APP p) : σ ⟩
step ⟨ Γ | (let • = x in y)^t | σ ⟩                 -- (Rule: let)
  = do let (p, heap') = Γ [⋆ ↦ (NonRec, x)]
       Just ⟨ heap' | y [• := p] | σ ⟩
step ⟨ Γ | (letrec • = x in y)^t | σ ⟩               -- (Rule: letrec)
  = do let (p, heap') = Γ [⋆ ↦ (Rec, x [• := p])]
       Just ⟨ heap' | y [• := p] | σ ⟩
step ⟨ Γ | (case x of as)^t | σ ⟩                    -- (Rule: case_0)
  = do Just ⟨ Γ | x | (t, CAS as) : σ ⟩
step ⟨ Γ | (\•→ x)^t | (_, APP p) : σ ⟩              -- (Rule: app_1λ)
  = do Just ⟨ Γ | x [• := p] | σ ⟩
step ⟨ Γ | (Con c ps)^t | (t1, APP p) : σ ⟩          -- (Rule: app_1c)
  = do Just ⟨ Γ | (Con c ((t1, Fre p) : ps))^t | σ ⟩
step ⟨ Γ | (Con c ps)^t | (_, CAS as) : σ ⟩          -- (Rule: case_1)
  = do y ← join $ listToMaybe
                  [ y [•... := [p | (_, Fre p) ← ps]]
                  | (c', y) ← as, c ≡ c' ]
       Just ⟨ Γ | y | σ ⟩
step ⟨ Γ | x | (_, UPD p) : σ ⟩                     -- (Rule: var_1)
  = do Just ⟨ (Γ [p ↦ x]) | x | σ ⟩
step (_)                                            -- (Rule: crash)
  = do Nothing


  -- Evaluate under operational semantics
run :: State t → Value t
run ⟨ Γ | (Con c ps)^t | [] ⟩ = Just (Γ, (Con c ps)^t)
run s                         = step s >>= run
```

Figure 5.4: Operational semantics for the core language.

```
  -- Normalise a state under the operational semantics
normalise :: State t → Maybe (State t)
normalise s = do s' ← step s
                 if isNormal s' then Just s' else normalise s'

  -- Determine if a state is in normal form
isNormal :: State t → Bool
isNormal ⟨ Γ | (Var (Fre p))^t | _ ⟩ = maybe True ((≡ Rec) ∘ fst) $
                                        lookup p Γ
isNormal s                           = isHalt s

  -- Determine if a state is halting
isHalt :: State t → Bool
isHalt ⟨ _ | (Con _ _)^t | [] ⟩     = True
isHalt ⟨ _ | (\•→ _)^t | [] ⟩       = True
isHalt ⟨ Γ | (Var (Fre p))^t | _ ⟩ = ¬ (p `isAllocated` Γ)
isHalt _                             = False
```

Figure 5.5: Normaliser for the core language.

recursion to ensure *strong normalisation*. Such a rule is only permitted if the termination criterion (Section 5.3.2) is *not* triggered.

Normalisation-by-evaluation is very attractive for the purposes of constructing a verifiable supercompiler, as it creates an explicit link between normalisation and the language semantics. Our normaliser, using the semantic *step* function, is shown in Figure 5.5. The *isNormal* function detects whether a state is in *normal form*. We define normal form to be any state that is a variable reference to a recursive binding or any halting state, detected by the *isHalt* function.

For example, consider the result of normalising the state created by applying *initialState* to our motivating example from Section 5.1.

```
 >>> normalise $ initialState example
Just
⟨ (HP 1,fromList [(HP 0,(Rec, \f xs ->
          case xs of { Nil       -> Nil;
                       Cons y ys -> Cons (f y) (map f ys) }
          in \f g xs -> map f (map g xs)))])
| ~HP 0~
| [] ⟩
```

The state is evaluated until the control expression is a variable reference to the heap location corresponding to `map`. As `map` is a recursive definition, this expression is in normal form.

```
  -- Record of tag appearances in abstract syntax tree.
type Bag t = Map t Int
type History t = [Bag (Tag t)]
  -- Record of where in the state the tag was found.
data Tag t = Heap t | Focus t | Stack t
  -- Ordering over bags.
(⊴) :: Ord t ⇒ Bag t → Bag t → Bool
ls ⊴ rs = Map.keysSet ls ≡ Map.keysSet rs ∧
          and [ maybe False (⩾ v) (Map.lookup k rs)
              | (k, v) ← Map.toList ls]
  -- Determine if normalisation should continue.
canContinue :: Ord t ⇒ History t → State t → Maybe (History t)
canContinue hist s | any (⊴b′) hist = Nothing
                   | otherwise      = Just (b′ : hist)
  where b′ = stateTags s
  -- Calculate a tag bag from a state.
stateTags :: Ord t ⇒ State t → Bag (Tag t)
```

Figure 5.6: Tag-based termination.

5.3.2 Terminator

Online termination of rewriting systems is a well-developed topic. A wide variety of termination conditions are available from and used in the literature (Leuschel, 2002). A popular technique is to detect *homeomorphic embeddings* of terms, appealing to Kruskal's (1960) *Tree Theorem*.

Mitchell (2010) takes a different approach where every node in the abstract syntax tree is *tagged* with elements from a finite set. Bolingbroke et al. (2011) supplies a library for constructing and experimenting with various termination conditions but settles on an implementation of Mitchell's technique. Bolingbroke (2013, chapter 6) further exploits the tagging technique in the splitter for term generalisation.

The tagging approach is better suited to *normalisation-by-evaluation* as no expression reconstruction is required. Every node in the source program's abstract syntax tree is decorated with a unique *tag*. The tags associated with syntactic elements are preserved by the operational semantics, migrating into the heap and the stack when appropriate. See Figure 5.4 for detail.

If, after normalisation, the abstract machine state is focused on a variable referring to a recursively defined expression, the terminator checks whether the bag of tags appearing in the state is *bigger* than any previously seen tag bags

under the $(\trianglelefteq)$ ordering. The relation $b \trianglelefteq b'$ holds iff: *(1)* the sets of tags in b and b' are equal and *(2)* every tag appears at least as frequently in b' as it does in b.

Our implementation, shown in Figure 5.6, distinguishes tags found in the state heap, focus and stack as suggested by Bolingbroke (2013, chapter 6). The *canContinue* function searches a historical record of tag bags and only returns a new history if the current state does not trigger the termination condition.

Consider the simplest non-terminating program that can be written in our core language: `(let x = x in x)`. We shall observe the sequence of states that result from its repeated normalisation and stepping. In this representation, we display tags as superscript integers.

```
 >>> initialState infloop
⟨ (HP 0,fromList [])
| (letrec x = x^(1) in x^(2))^(0)
| [] ⟩

 >>> fromJust $ normalise it
⟨ (HP 1,fromList [(HP 0,(True,~0~^(1)))])
| ~0~^(2)
| [] ⟩

 >>> fromJust $ normalise it
⟨ (HP 1,fromList [(HP 0,(True,~0~^(1)))])
| ~0~^(1)
| [(^(2),UPD (HP 0))] ⟩

 >>> fromJust $ normalise it
⟨ (HP 1,fromList [(HP 0,(True,~0~^(1)))])
| ~0~^(1)
| [(^(1),UPD (HP 0)),(^(2),UPD (HP 0))] ⟩
```

On the final normalisation, the termination condition would be met as all the tags in the third intermediate state appear in the same state components, and appear at least as often, in the fourth intermediate state. This is indicative of source program recursion that could be normalised forever as the same syntactic elements are being added to the state.

From a verification perspective, the terminator never modifies the source state. Instead, it is used to control supercompiler execution flow. It should not interfere with *semantic correctness* proofs. Verification of termination properties of a compiler are often a low priority. For example, Mitchell (2010) explicitly highlights some of the edge cases for which the supercompiler he defines will not terminate.

```
data Split a b = Split {context :: Tree a → a, subterms :: Tree b}
data Tree a = Leaf a | Branch [Tree a]

mkIdentity :: a → Split a b
mkIdentity x = Split (const x) (Branch [])

mkLeaf :: b → Split a b
mkLeaf x = Split (λ(Leaf x) → x) (Leaf x)

mkBranch :: ([a] → a) → [Split a b] → Split a b
mkBranch f xs = Split
  (λ(Branch ys) → f $ zipWith (($) ∘ context) xs ys)
  (Branch $ map subterms xs)
```

Figure 5.7: Data types representing splitting notation.

5.3.3 Splitting

Once a term has been normalised, we must *split* the term into smaller subterms on which supercompilation continues. Supercompilers that operate by rewriting expressions split syntactically, often generalising with respect to previously seen expressions. Supercompilers that operate through normalisation-by-evaluation split the normalised states into smaller states and then recombine the results of supercompilation as expressions. Bolingbroke and Peyton Jones (2010) state that *"a good split function will residualise as little of the input as possible, [further optimising] as much as possible"*. While not necessary to preserve semantic correctness, a good split function must also conserve sharing.

The splitting process is described using the terminology of *subterms* which represent the isolatable components of the current state and *contexts* into which we shall insert the results of supercompiling the subterms. This is reified as a data type in Figure 5.7.

The splitting algorithm, introduced in Figure 5.8, is a pipeline of three phases, corresponding to the three components of an abstract machine state: *(1)* focus expression splitting, *(2)* control stack splitting and *(3)* heap splitting. Each phase, where appropriate, maintains lists of heap locations that may lose sharing through non-linear use or missed updates.

SPLITTING FOCUS EXPRESSIONS There are four possible circumstances under which the splitting algorithm may be called: *(1)* a variable references a `letrec` binding and the termination condition is triggered; *(2)* a lambda is returned as a value; *(3)* a variable references an *'unknown'* value; *(4)* a construction is returned as a value. These four cases correspond to different equations in the definition of *focusSplit* in Figure 5.9:

```
split :: Show t ⇒ State t → HeapSplit t
split s = heapSplit (updateMany upds Γ) freeRefs app splitStack
  where ⟨ Γ | x | σ ⟩ = moveArgs s
        (freeRefs, splitFocus) = focusSplit Γ x
        (splitStack, (app, upds)) = runWriter (stackSplit σ splitFocus)

moveArgs :: State t → State t
moveArgs ⟨ Γ | (Con c ps)^t | σ ⟩
  = ⟨ Γ | (Con c [])^t | foldl (λstk (t, Fre p) → (t, APP p) : stk) σ ps ⟩
moveArgs s = s

  -- Apply many updates to a Heap
updateMany :: Upds t → Heap t → Heap t

  -- Specialised intermediate splits
type FocusSplit t = Split (Scopes t) (Scopes t)
type StackSplit t = Split (Scopes t) (Scopes t, Stack t)

type FreeRefs = Set HP
type Apps     = Set HP
type Upds t   = Map HP (Exp t)
```

Figure 5.8: Splitting a normalised state.

```
focusSplit :: Heap t → Exp t → (FreeRefs, FocusSplit t)
focusSplit Γ x = case x of
  (Var (Fre p))^t | isAllocated p Γ
                  → (Set.singleton p, mkLeaf (Closed x))
  (\•→ (Scope y))^t → (freeExp y,       lambdaSplit t y)
  _               → (Set.empty,       mkIdentity (Closed x))

lambdaSplit :: t → Exp t → FocusSplit t
lambdaSplit t body = Split (λ(Leaf (Open (Closed body))) →
                              Closed (\•→ (Scope body))^t)
                           (Leaf (Open (Closed body)))
```

Figure 5.9: Splitting the focus component of states.

```
stackSplit :: Stack t → FocusSplit t → Writer (Apps, Upds t) (StackSplit t)
stackSplit []                    focus = do
  return $ addEmptyStack focus
stackSplit ((t, APP p) : σ)  focus = do
  tell (Set.singleton p, mempty)
  stackSplit σ $ mkBranch (λ[Closed x] → Closed (x ⊔ (Fre p))^t) [focus]
stackSplit ((t, UPD p) : σ)  focus = do
  tell (mempty, Map.singleton p (reformFocus focus))
  stackSplit σ $ mkLeaf (Closed (Var (Fre p))^t)
stackSplit ((t, CAS as) : σ) focus = do
  return $ mkBranch
    (λ(Closed x : ys) → Closed (case x of (zip (map fst as) ys))^t)
    (addEmptyStack focus : [mkLeaf (y, σ) | (c, y) ← as])

addEmptyStack :: FocusSplit t → StackSplit t
addEmptyStack = fmap (, [])

reformFocus :: FocusSplit t → Exp t
reformFocus (Split ctx stm) = stripScope $ ctx stm
  where stripScope (Closed x) = x
        stripScope (Open x) = stripScope x
```

Figure 5.10: Splitting the control stack component of states.

1. A variable reference that triggers the termination condition creates a subterm independent of control stack so that it can be supercompiled in isolation.

2. When a lambda is returned as a value, the binding is the context and the body is the subterm.

3. A reference to an unknown can be supercompiled no further. The *context* is the reference itself and there are no *subterms*.

4. The constructor splits with each component argument as a subterm and the spine as the context. This is achieved by placing the arguments back on the control stack using *moveArgs* before following the same logic as in (3).

SPLITTING THE CONTROL STACK Once useful information is extracted from the focus expression, the algorithm splits the control stack as shown Figure 5.10. It is essential to preserve sharing and partially supercompiled results. So the algorithm records the heap pointers that are being independently supercompiled

and any incomplete heap update operations. These pointers are stored through a *Writer monad* and will be used in the final *heap splitting* stage.

- Application stack frames, of the form *APP p*, result in the context being extended with the application of the heap pointer *p*. The heap location *p* is recorded for further supercompilation.
- Update frames, *UPD p*, result in the current context being reformed into an expression and stored at heap location *p*. The context becomes a variable pointing to heap location *p*.
- Case distinction frames, *CAS as*, produce subterms for each case alternative in *as* with the remaining control stack.

SPLITTING THE HEAP At this stage, any updates are applied to the heap, so that partial supercompilation results can be shared. Then we use the application and update information provided by the previous stages to determine which heap locations are *'unsafe'* — at risk of losing sharing. Any heap location that may lose sharing is extracted and supercompiled independently. All these steps can be seen in the definition of *heapSplit* in Figure 5.11.

5.3.4 Memoisation

Given the presence of recursion in our source language, an execution may be non-terminating. Furthermore, there is a combination of recursion *and* unknown values in the context of supercompilation. We risk replicating previously supercompiled results *unless* these results are *memoised* and shared.

The memoisation component works in tandem with the termination component. The terminator detects the possibility of repeated work, then the memoiser creates a reference to that work.

In this implementation, each chain of normalisations is labelled with a *binder*, a fresh heap pointer and record of the unknown variables introduced due to lambdas. This behaviour is shown in the *mkBinder* function of Figure 5.12. Each state normalisation is recorded with the label of the corresponding chain as a *promise* of a terminating supercompilation. When the memoisation algorithm, *memoise*, detects that the current state is equivalent to a previous promise, an reference to the corresponding binder is constructed.

5.3.5 Control

An overarching controller algorithm marshals these components to achieve supercompilation. As shown in Figure 5.13, the *supercompile* function accepts a tagged expression as input and returns an untagged expression. It begins by

```
type HeapSplit t = Split (Exp ()) (State t)
heapSplit :: Heap t → FreeRefs → Apps → StackSplit t → HeapSplit t
heapSplit Γ freeInFocus app focus = foldr aux focus' unsafe
  where
    -- Function to calculate accessible heap pointers
    accessibleRef = fixpointRef Γ
    -- Heap pointers to be driven independently
    unsafe        = Set.filter (`isAllocated`Γ) $
                    Set.union app $ intersections $ map accessibleRef $
                    freeInFocus : map Set.singleton (Set.toList app)
    -- Heap without unsafe locations
    safeHeap      = removeMany unsafe Γ
    focus'        = addHeap safeHeap focus
    -- Create a subterm for heap location p
    aux p spl     = mkBranch letIn [mkLeaf ⟨ heap' | x^t | [] ⟩, spl]
      where
        Just (rec, x^t) = lookup p Γ
        heap' = copy p Γ safeHeap
        letIn [x, y] | rec ≡ Rec = (letrec • = x [p := •] in y [p := •])^()
                     | otherwise = (let • =     x in         y [p := •])^()
  -- Add a heap to a split
addHeap :: Heap t → StackSplit t → HeapSplit t
  -- Calculate the union of every intersection between distinct sets
intersections :: Ord a ⇒ [Set a] → Set a
  -- Assume a closed term
fromClosed :: Scopes t → Exp t
  -- Bind all dangling references to fresh heap pointers
bindOpen :: Heap t → Scopes t → (Heap t, Exp t, [HP])
  -- Remove many pointers from a Heap
removeMany :: Set HP → Heap t → Heap t
  -- Copy from one heap to another
copy :: HP → Heap t → Heap t → Heap t
```

Figure 5.11: Splitting the heap component of states.

```
type Memo a = (Binder, [Promise]) → a
type Binder  = (HP, Set HP)
type Promise = (Binder, State Int)

type Phase = State Int → Memo (Exp ())

mkBinder :: Phase → Phase
mkBinder cont s ((i, _), prevs)
  | i' ∈ freeExp x' = (letrec • = x' [i' := •] in Scope (Var •)^()))^()
  | otherwise       = x'
  where i' = pred i
        (ps', s') = stripLambdas s
        x' = reconstructLambdas ps' (cont s' ((i', ps'), prevs))

memoise :: Phase → Phase
memoise cont s (this, prevs) | null matches = cont s (this, (this, s) : prevs)
                             | otherwise    = head matches
  where matches = [ foldr aux (Var (Fre iOld))^() freeOld
                  | ((iOld, freeOld), sOld) ← prevs
                  , freeMapping ← maybeToList (sOld `equivState` s)
                  , Map.keysSet freeMapping ≡ freeOld
                  , let find p = Map.findWithDefault p p freeMapping
                  , let aux p x = (x ⊔ (Fre (find p)))^() ]

  -- Strip lambdas and instantiate with fresh heap pointers.
stripLambdas :: State t → (Set HP, State t)

  -- Reform lambdas, abstracting given heap pointers.
reconstructLambdas :: Set HP → Exp t → Exp t

  -- Test if two states are alpha-equivalent, returning the mapping.
equivState :: (Data a, Data b) ⇒ State a → State b → Maybe (Map HP HP)
```

Figure 5.12: Memoising results.

```
supercompile :: Exp Int → Exp ()
supercompile x = mkBinder (drive []) (initialState x) ((HP 0, Set.empty), [])

drive, drive' :: History Int → Phase
drive  hist   = memoise (drive' hist)
drive' hist s = case (isHalt s, normalise s) of
  (False, Just s') → maybe (memoise $ tie hist) drive (canContinue hist s) s'
  otherwise        → tie hist s

tie :: History Int → Phase
tie hist s memo = ctx (fmap residualise stms)
  where Split ctx stms = split s
        residualise s = mkBinder (drive hist) s memo
```

Figure 5.13: Control algorithm.

constructing a promise for a state representation of the expression and *driving* with an empty history.

A memoised *drive* function tests if the input state is already halting or if normalising it further results in a crash. Either of these conditions result in the residualisation of subterms using the *tie* function.

Alternatively, the input state is tested against the termination condition. If this condition is not met, the state is driven further. If the termination condition is met, the subterms are residualised using a memoised variant of the *tie* function. The recursive interaction of *drive* and *tie* on subterms results in the construction of a *residual program*, the goal of supercompilation.

5.4 DISCUSSION

This chapter has introduced the key concepts in supercompilation and provided a reference implementation. This represents the first stage towards our goal of producing a verified supercompiler and we should analyse what we have seen so far.

SEMANTIC TYPES The normalisation, splitting and memoisation components all operate on types for which we can define a semantics. Normalisation takes states to states. Memoisation from states to expressions in some context. Splitting takes states and produces a collection of substates and a method of reconstructing an expression.

Given that these are the only functions that construct the residual program, if these functions maintain semantic correctness locally, then global semantic correctness should follow. This will be investigated further in Chapter 7.

CONTROL FLOW AND TERMINATION The control algorithm, discussed in Section 5.3.5, has a control flow based on mutually recursive calls, often through higher-order functions. Any change to the control algorithm will tend to affect the termination of the supercompiler rather than change the semantics of the resulting residual programs. However, its recursive higher-order nature does complicate the process of verification. This issue reappears in Section 8.6.

A total correctness proof of termination will require an encoding of the pigeonhole principle and demonstrating a well-founded decreasing ordering over states through the recursive calls. Many mechanical proof system have their own internal representations of program termination (for example Abel, 2000) and we would need to fit supercompiler termination to these models.

COMPONENT COMPLEXITY Some components would be easier to verify with mechanised proof than others. For example, showing a semantic link for the normalisation component is considerably easier (due to its close relationship with the operational semantics) than for splitting. A hybrid verification model could take advantage of this, using lightweight verification methods, such as automated testing, for some components and combining it with heavyweight verification, such as mechanised proof, for others. We shall require a hybrid verification framework that enables modularity in the verification of different components.

5.5 SUMMARY

In this chapter, we have implemented a supercompiler for a small functional language. Through this implementation, we have introduced common terminology and discussed design choices made in other implementations, such as those from Bolingbroke (2013); Bolingbroke and Peyton Jones (2010); Jonsson and Nordlander (2008); Klyuchnikov (2010); Mitchell (2010); Mitchell and Runciman (2008); Reich et al. (2010); Sørensen et al. (1996).

A number of themes emerge when considering supercompilation for verification. *(1)* We can define properties about the semantic nature of components. *(2)* The control flow of the program presents challenges in decomposing the problem for proof. *(3)* Proof of termination will be difficult given the constructs of the program. *(4)* Some form of hybrid verification will probably be necessary.

```
  -- Instantiate an open term with a single heap pointer
· [• := ·] :: HP → Scope t → Exp t
  -- Instantiate an open term with multiple heap pointers
· [•... := ·] :: [HP] → Scopes t → Maybe (Exp t)
  -- Initial abstract machine state
initialState :: Exp t → State t
initialState x = ⟨ (HP 0, Map.empty) | x | [] ⟩
  -- Store on heap, allocating a fresh heap pointer
· [⋆ ↦ ·] :: (IsRec, Exp t) → Heap t → (HP, Heap t)
(nxt, Γ) [⋆ ↦ x] = (nxt, (succ nxt, Map.insert nxt x Γ))
  -- Lookup on the heap
lookup :: HP → Heap t → Maybe (IsRec, Exp t)
lookup p (_, Γ) = Map.lookup p Γ
  -- Check if a heap pointer is in use
isAllocated :: HP → Heap t → Bool
isAllocated p Γ = isJust (lookup p Γ)
  -- Update the expression at a heap location.
· [· ↦ ·] :: HP → Exp t → Heap t → Heap t
(nxt, Γ) [p ↦ x] = (nxt, Map.adjust (second $ const x) p Γ)
```

Figure 5.14: Auxiliary functions for the operational semantics.

6 COMPILER VERIFICATION THROUGH PROOF

This chapter surveys some of the literature in the field of compiler verification. We discuss some of the seminal publications, highlight some of the techniques used to structure compiler verification and consider some of the tools used to aid the verification process. Our findings will inform the hybrid verification of a supercompiler.

6.1 THE ORIGINS OF COMPILER CORRECTNESS

The earliest example of compiler verification is reported by McCarthy and Painter (1967). In their seminal paper, they produce a hand proof of correctness for a compiler that translates a simple source arithmetic language to instructions for a register machine.

The source language only contains natural numbers, variables bound to natural numbers and the binary addition operator. The target language has just four instructions:

- `LI n` — Load immediate value into the accumulator.
- `LOAD r` — Load the value of register r into the accumulator.
- `STO r` — Store the value of the accumulator in register r.
- `ADD r` — Add the value of register r to the accumulator.

McCarthy and Painter encode the semantics of the two languages into a *first-order predicate logic*. States for the source semantics (and the target semantics) are defined as predicates over variables (or registers) containing natural number values.

A compiler is also defined in terms of first-order predicate logic and, as an argument, takes a function that maps variables bound in the source state to registers initialised in the target state.

A relation of *partial equality* between states is defined. For states σ_1 and σ_2 and a set, A, of variable names, the relation $\sigma_1 =_A \sigma_2$ holds if $\forall x \in A \cdot \sigma_1(x) = \sigma_2(x)$. Using this condition, McCarthy and Painter construct a theorem for compiler correctness, which is proved using induction over the source language and the lemmas about register access, that they had previously defined.

The authors point out that it is trivial to extend the proof to handle a source language that contains multiplication. However, constructs such as sequential

composition, conditionals and jump statements would require *"a complete revision of the formalism"* (McCarthy and Painter, 1967).

By modern standards, the proof appears verbose. It exposes a large amount of the theory that is taken for granted in a modern logic for reasoning about computer languages. Although the source language may not contain the features required for it to be considered 'useful' in a software engineering context, the McCarthy and Painter (1967) example posed a vital question and laid the foundations for a field.

6.2 MECHANISED PROOF

By 1972, there was work investigating the use of proof assistants and mechanised logics to prove correctness properties of compilers. Using LCF (Milner, 1972), an implementation of Scott's Logic of Computable Functions, Milner and Weyhrauch prove the example from McCarthy and Painter (1967) and begin to demonstrate a machine checkable proof for an *"ALGOL-like language with assignments, conditionals, whiles and compound statements"* (Milner and Weyhrauch, 1972).

Milner and Weyhrauch (1972) cite the work of other researchers who have either mechanised proofs for languages of about the same complexity as the one used by McCarthy and Painter (1967) or else have accepted hand proofs of correctness for ALGOL and LISP compilers. The contribution of Milner and Weyhrauch (1972) involves combining the two goals of machine-checkable proof and the correctness of a more substantial compiler. They also use a different formulation of the constructs involved, based on the work of Lockwood Morris (1973).

Instead of the first-order predicate logic used by McCarthy and Painter, Milner and Weyhrauch used a definition of the semantics based in *"the theory of typed lambda calculus, augmented by a powerful induction rule"*. The *"meaning of the program is a partial computable function from states to states"* (Milner and Weyhrauch, 1972), the denotational semantics for a language. Furthermore, states themselves are represented as functions from names to values. Many of the low-level laws that must be explicitly stated for programs declared in first-order predicate calculus are implicit in typed lambda calculus, leading to smaller, more manageable proofs.

The proof is summarised in Figure 6.1. S and T are the source and target languages. MS and MT are the semantics of those languages expressed as partial functions from states to states. $\hat{S}$ and $\hat{T}$ are state functions, mapping names to values. The compiler function, $comp$, takes source programs to target code. The proof is that a retrieval function, $SIMUL$ can be constructed to complete the diagram.

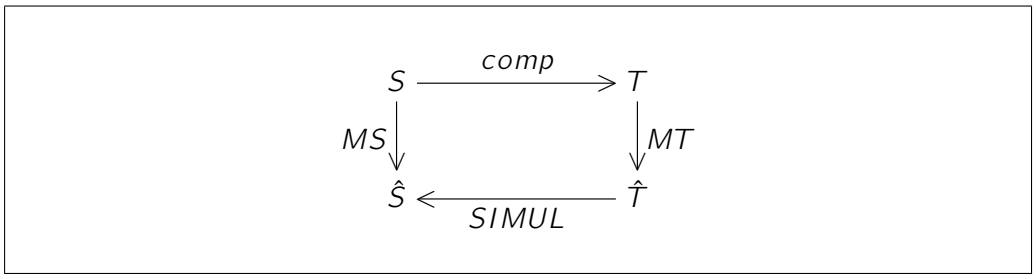

Figure 6.1: Commutative diagram from Milner and Weyhrauch (1972)

The paper demonstrates a complete, machine-checked proof of the McCarthy and Painter compiler using a lambda calculus encoding. The further goal of machine-checked proof for a compiler for an ALGOL-like language is left incomplete, but Milner and Weyhrauch have *"no significant doubt that the remainder of the proof can be done on the machine."*

Although they remain uncommitted about the algebraic approach to their proof, Milner and Weyhrauch (1972) are of the opinion that *"for machine-checked compiler proofs some way of structuring the proof is desirable"*.

6.3 ALGEBRAIC MODELS

Milner and Weyhrauch acknowledge that their algebraic approach is inspired by discussions with Lockwood Morris. He develops the details of this algebraic method in the subsequently published paper, *'Advice on structuring compilers and proving them correct'* (Lockwood Morris, 1973).

The *"essence"* of the advice presented in the paper consists of the commuting diagram, Figure 6.1. Although different notation is used, the diagram conveys the same information.

However, Lockwood Morris explicitly discusses the algebraic interpretation of the diagram's components. The languages, S and T, and meanings, $\hat{S}$ and $\hat{T}$, are described as *"heterogeneous (universal) algebras,"* much like algebraic data types in modern languages. In particular, S and T are *initial algebras*, such that there is a *unique homomorphism* to any other algebra.

The semantics MS and MT are mappings of respective languages to meanings. The functions $comp$ and $SIMUL$ are mappings between languages and between meanings, respectively. These, under the algebraic interpretation, become *homomorphisms*.

For an example, we shall discuss a homomorphism between two data types. Consider algebraic data types A and B. These each have one or more constructors which may take any number of arguments. For example, C_0 is a constructor

in the A data type that takes $(m, 0)$ arguments of type A. A homomorphism, ϕ from A to B would (for all x and some y) satisfy Equation 6.1.

$$\phi(C_x(z_0, z_1, ..., z_{(m,x)})) = D_y(\phi(z_0), \phi(z_1), \phi(z_{(m,x)})) \tag{6.1}$$

The very definitions of MS, MT and $comp$ in Morris's examples show that they are homomorphisms. Due to the initiality of the S and T languages, the $comp$, MS and MT functions are *unique homomorphisms*. All that remains is to show that $encode$ is a homomorphism, where $encode$ is the inverse of $SIMUL$. Once this is confirmed, by the *unique extension lemma*, Equation 6.2 must be true, as there is only one homomorphism from an initial algebra to any other.

$$encode \circ MS = MT \circ comp \tag{6.2}$$

The example compiler in Lockwood Morris (1973) compiles a small, ALGOL-like language to a flowchart representation of a stack machine. However, the proof is left incomplete.

Thatcher et al. (1980) reasons that the difficulty is largely the lack of algebraic structure on the right-hand side of the commuting diagram. Their paper, '*More on advice on structuring compilers and proving them correct*', extends the Lockwood Morris (1973) work by enlarging the source language further and by introducing a *categorical* representation of flowcharts over a stack machine as the target language. Most importantly, Thatcher et al. (1980) complete the proof, where Lockwood Morris (1973) could not, by the same unique extension principle.

Thatcher et al. (1980) commend the *"extremely powerful methodology"* as *"no structural induction is required for the definition of the arrows or the proof."* However, they admit that the proofs that $encode$ is a homomorphism are *"considerably longer and more cumbersome than [...] expected."*

One wonders whether this is a problem with their representations, notation or an issue with the general algebraic technique. Furthermore, does this imply that the proofs will become intractable as larger languages are introduced?

6.4 DECOMPOSITION OF COMPILERS

Meijer (1994) takes a different view of compiler correctness. Instead of building a compiler and proving it correct, why not *"improve a correct compiler?"*

The denotational semantics of the source language, MS in Figure 6.1, can be expressed as a functional program that takes a source program and produces a function mapping source states to values. Can we decompose this function to produce the other functions ($comp$, MT and $SIMUL$) in the commuting diagrams?

Meijer (1994) demonstrates such a decomposition for a first-order imperative source language being compiled to a three-address code register machine. He

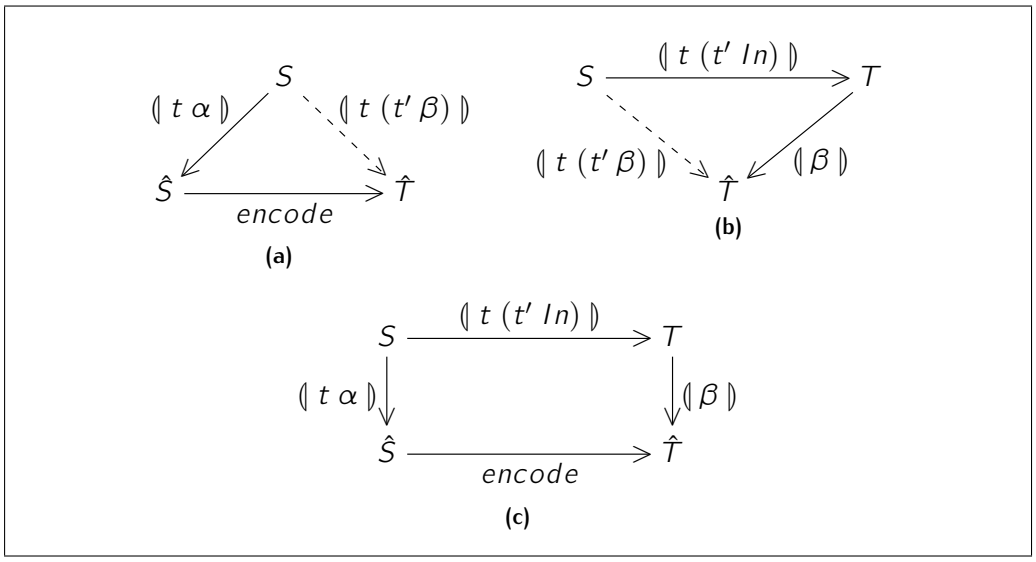

Figure 6.2: From Meijer (1994). Diagrams (a) and (b) represent the divided correctness problem. (c) is the combination of these triangles to give the usual correctness commutativity diagram.

appeals to the same algebraic concepts of Lockwood Morris (1973) and Thatcher et al. (1980).

The commuting diagram is split as shown in Figure 6.2. Notable differences from previous work include the explicit realisation of various homomorphisms as *catamorphisms*, signified by the *'banana brackets.'* Catamorphisms have been referred to in the previous literature as *unique homomorphisms* but were functions constructed to fit the homomorphic structure. Meijer (1994) instead produces functions that can be expressed using a standard catamorphic operator, ensuring their catamorphic (and homomorphic) properties. In this sense, we can think of them as a generalisation of the functional *fold* operator (Meijer et al., 1991).

Meijer (1994) formally transforms the catamorphic form of the source semantics into the other homomorphisms in the diagram. In this way, he succeeds in producing a compiler that is correct with respect to the rest of the diagram.

This technique seems particularly suited to first-order languages. *"[Things] get notoriously hard when the domains themselves become recursive, especially when function spaces are involved"* (Meijer, 1994). So, this technique may not suitable for proofs about dependable compilers for a higher-order functional language.

6.5 EQUIVALENT GRAPHS

Lazy functional languages can benefit from concise definitions through high-level abstractions. Programs in these languages are often compiled for graph-reduction machines, of which the *G-machine* (Augustsson, 1984) is the most

widely studied. In his thesis, Lester (1988) proves that compiling a lazy functional language to a G-machine implementation is correct.

Lester (1988) defines the denotational semantics *and* the operational semantics of the lazy functional language. The operational semantics are modelled as G-machine reductions.

In terms of the previous commuting diagram (Figure 6.1), the denotational semantics still represent the MS morphism. However, the operational semantics represent the composition of MT and $comp$, with the target meanings, $\hat{T}$, at a higher level of abstraction to the state machine specifications used in previous work.

Interestingly, there is a return to the decoding of target meanings into source meanings, rather than the inverse used by Lockwood Morris (1973), Thatcher et al. (1980) and Meijer (1994). The reasons why a decoding (concrete to abstract state) relation is used, rather than an encoding (abstract to concrete state) relation is not documented. It may be because it is more natural to consider the retrieval of a less-defined state from a more-defined one.

Lester (1988) begins with a proof for a small lazy functional language that only accounts for programs that terminate under the source semantics. The proof makes use of fixpoint induction over states and structural induction of the abstract syntax representations of the source language. Similar techniques are then used to extend the proof to include built-in functions, data structures and some compiler optimisations.

6.6 FUNCTIONAL ABSTRACTIONS

Mintchev (1995), as part of his thesis, develops a mechanised version of the hand proof by Lester (1988), his supervisor. Proofs are developed in Isabelle/LCF and then again in a theorem prover that is developed as part of the thesis. Mintchev's theorem prover operates on a subset of the Haskell language, dubbed *Core*.

During the development of an Isabelle/LCF proof, an error was found in the original hand proof by Lester (1988). This discovery highlights the benefits of using proof assistants for checking theorem validity. However, the recursive nature of the source language is highlighted as a source of difficulty for the logic, requiring the proof of a highly technical domain theoretic lemma *"taking up 85% of over 2000 lines of definitions, axioms, theorems and tactics"* (Mintchev, 1995).

Mintchev (1995) uses his own theorem prover to develop a proof for a variation of the problem tackled in Lester (1988). The operational semantics are for a *spineless* G-machine (Burn et al., 1988) as *"it avoids unnecessary updates after each reduction step"* (Mintchev, 1995).

Another distinctive aspect of Mintchev's work is the use of *monads*, now a common functional programming abstraction, to describe the *"heap free operational semantics."* Mintchev (1995) states that monads were originally used as useful abstraction for defining the abstract machine but *"paid off in the proof of correctness."*

The thesis does not discuss facilities for checking the soundness of Mintchev's custom theorem prover. Without verifying the theorem prover itself, no guarantees can be made that the proofs it produces are valid.

6.7 PROGRAM EXTRACTION

One of the perceived advantages of the Mintchev (1995) theorem prover was that it operated directly on program code, maintaining the direct link between theorems and the executable object.

Berghofer and Strecker (2004) investigate using Isabelle/HOL to verify a compiler for a simplified, Java-like language, $\mu Java$. In their approach, they use Isabelle's code extraction facility to produce an ML program.

In addition to verifying the semantic equivalence of compiled programs, they also prove a number of other compiler functions correct such as type-checking and pretty printing. The result that the program is well-typed is a necessary precondition for the correctness of code translation.

Berghofer and Strecker do not say much about how they structure the compiler to aid the verification process. Instead, they focus on the correctness conditions that are verified and the preconditions that these depend on.

Berghofer and Strecker (2004) do question the dependability of the Isabelle code extraction facility as it is *"a complex piece of code and thus prone to errors."* There is little literature on what guarantees have been developed about the correctness of Isabelle code extraction since 2004.

6.8 PRACTICAL COMPILERS

Leroy (2009) represents a monumental achievement in the field of compiler verification, the result of two man-years of effort. The artefact produced is a mechanically verified compiler implementation which translates a C-like language, Cminor, to PowerPC assembly code.

The compiler is developed and proved correct using the Coq proof assistant. Similar to Berghofer and Strecker (2004), a code extraction tool processes the theorems to produce an executable Caml program. Unlike the Isabelle/HOL extraction facility (circa 2004), the Coq code extractor appears to be held in higher esteem (Letouzey, 2008). Even so, Leroy (2009) indicates that one of their

research aims is to investigate the *"feasibility of formally verifying Coq's extraction mechanism and a compiler from Mini-ML to Cminor."*

The correctness condition used by Leroy (2009) is as follows. For all source programs, if compilation completes successfully and the source program evaluates to a non-error value, then the compiled program should evaluate to the same value. This condition is valid for the compiler that fails for all input. Leroy (2009) states that *"whether the compiler succeeds to compile the source programs of interest is not a correctness issue, but a quality of implementation issue, which is addressed by non-formal methods such as testing."*

An essential lemma is that if any two compilers are verified, then their composition is also verified. This enables the process of developing a verified compiler to be split into several independent proof efforts, which can easily be composed to form a proof for an entire compiler.

One highlighted area of weakness is *"do the formal semantics of Cminor and PPC, along with the underlying memory model, capture the intended behaviours"* (Leroy, 2009)? As languages of increasing complexity are verified, it may be more difficult to maintain confidence in the initial specifications. Possible solutions include: *(a)* ensuring that the language specification remains small enough that it can be checked successfully, *(b)* using a variety of semantic representations simultaneously and checking for inconsistency, or *(c)* generating the lower-level virtual or hardware machines from semantic specifications.

The only other concern that Leroy (2009) raises is that the Coq logic or implementation might not be sound. This is highly unlikely as the kernel of Coq has been kept intentionally small. Furthermore, *"proofs mechanically checked ... are orders of magnitude more trustworthy than even carefully hand-checked mathematical proofs"* (Leroy, 2009).

6.9 SUPERCOMPILER CORRECTNESS

Some components of supercompilation correctness are founded in other mathematical proofs. For example, the proof that driving must terminate under homeomorphic embedding is derived from *Kruskall's Tree Theorem* (Kruskal, 1960).

A number of hand-proofs of correctness have been produced for entire supercompiler designs. According to Klyuchnikov (2010), prior to his own verification of the HSOC supercompiler, Sørensen et al. (1996) and Jonsson and Nordlander (2008) had published proofs of correctness for their respective supercompilers.

However, none of these proofs has been mechanised. Correspondingly, no automatically generated implementation has been produced. It is uncertain

how easily the manual proofs could be machine checked as, in many places there is appeal without proof to informally stated lemmas.

Krustev (2010) discusses *"a simple supercompiler formally verified in Coq."* The abstract highlights particular features that made the verification possible. *"First, a very limited object language; second, decomposing the supercompilation process into many sub-transformations, whose correctness can be checked independently"* (Krustev, 2010).

In a recent publication, Krustev (2013) builds on this result to introduce a *"a language-agnostic framework for building verified supercompilers,"* also defined in Coq. The model presented in Chapter 7 of this thesis was developed independently at around the same time. The two models share key characteristics: the decomposition of supercompilation into individually verifiable components and the use of language-independence as a simplifying assumption.

6.10 SUMMARY

Since McCarthy and Painter (1967), the formal verification of compilers has been an active topic due to its impact on the dependability of other compiled software. This survey only accounts for a sample of the available literature on the topic of formal reasoning about compilers. The aim has been to focus on research that uses algebraic abstractions to structure proofs or that uses mechanised tools to produce dependable compiler software.

Proof assistants not only help to ensure that a proof is sound. They are essential in managing the complexity of larger language specifications and more advanced compiler designs (Krustev, 2010, 2013; Milner and Weyhrauch, 1972; Mintchev, 1995). Furthermore, many modern proof assistants provide mechanisms for extracting executable programs from theorems, increasing confidence that the resulting software is a valid implementation of the formal specification (Berghofer and Strecker, 2004; Leroy, 2009).

Induction over source language or states is a common proof method (Lester, 1988; McCarthy and Painter, 1967). Alternatively, mathematical abstractions, such as initial algebras and unique homomorphisms, are used to reduce the proof by appealing to existing theorems (Lockwood Morris, 1973; Thatcher et al., 1980).

Meijer (1994) suggested using these abstractions to transform the source semantics into the necessary compiler. The proof would be apparent from the formal transformation process. This technique appears not to have been widely adopted due to the difficulty in applying it to higher-order languages.

Some hand-proofs have been completed for certain supercompiler designs (Jonsson and Nordlander, 2008; Klyuchnikov, 2010; Sørensen et al., 1996). Mech-

anised proofs exist for simple and abstract supercompilers (Krustev, 2010, 2013) but as of yet there is no complete mechanised proof of a supercompiler.

It is clear that proofs about compilers, whether by hand or with mechanisation, are costly to produce. It is unclear how proofs can be reused, and there is little discussion of how they could be performed incrementally, to fit in with an engineering process.

7 PROOF OF AN ABSTRACT SUPERCOMPILER

This chapter presents an encoding of an abstract supercompiler in Agda, a dependently-typed language. The key area of abstraction is over what source language is used. The abstract supercompiler provides a framework for modular proofs of correctness and will be incorporated into a hybrid verification model.

7.1 INTRODUCTION

Before verifying a particular supercompiler implementation, it is useful to distill the essence of what a supercompiler is and what it means for one to be correct. For this purpose, we shall define an *abstract supercompiler* in Agda (Norell, 2009), a dependently-typed programming language, encoding correctness properties as types and correctness proofs as values inhabiting those types.

The abstract supercompiler does not define a particular language syntax or semantics. Neither does it define a particular termination condition. Instead, it presents interfaces for these particular artefacts and exposes the minimal properties that should hold over them for a supercompiler that maintains *semantic correctness*.

Section 7.2 gives a brief overview of Agda's syntax and functionality, discussing how to encode properties and proofs in Agda's type system. Section 7.3 begins our exploration of an abstract supercompiler by characterising the types involved in an operational semantics. Section 7.4 encodes the concept of semantics preserving functions and demonstrates how such functions may be composed. Section 7.5 uses these constructs to describe a correct-by-construction supercompiler. Section 7.6 briefly discusses some issues with Agda's code extraction mechanism, MAlonzo, and motivates the approach that is taken in Chapter 8.

7.2 A WHIRLWIND INTRODUCTION TO AGDA

Agda (Norell, 2009) has a very similar syntax to Haskell, our usual host language, while permitting the expression of a wider range of properties than Haskell's type system. Critically, as a *dependently-typed* language, type specifications can *depend* on the values. We shall illustrate Agda's syntax and capability by example.

```
module Whirlwind where
  infixr 5 _::_ _++_
  data List (α : Set) : Set where
    []  :                          List α
    _::_ : (x : α) (xs : List α) → List α

  _++_ : ∀ {α} → List α → List α → List α
  []       ++ ys = ys
  (x :: xs) ++ ys = x :: (xs ++ ys)

  map : ∀ {α β} → (α → β) → List α → List β
  map f []       = []
  map f (x :: xs) = f x :: map f xs

  _∘_ : ∀ {α β γ : Set} → (β → γ) → (α → β) → (α → γ)
  (f ∘ g) x = f (g x)
```

Figure 7.1: Functional programming in Agda.

7.2.1 Functional programming

Figure 7.1 shows some basic functional programming concepts defined using Agda. This fragment defines a list data type, a concatenation operator, a list `map` function and a functional composition operator.

We declare a new module called `Whirlwind` in which we make our tutorial definitions. List α is an *algebraic data type* representing a polymorphic list data structure, equivalent to the list data type in Haskell. Notice that in Agda, the polymorphic parameter α and List α are both members of type **Set**. In Haskell 2010, data types belong to the kind $*$, where kinds are distinct from types.

The **infix** declaration should be familiar to Haskell users but Agda's parser also permits arbitrary mixfix operators, composed of any unreserved unicode characters. Therefore, one declares the fixity of an operator by using underscores to denote argument positions.

In Haskell 2010, type variables are implicitly declared and are implicitly bound by type-inference. In Agda all type variables must be explicitly declared although many may be implicitly bound by type-inference. These *implicit parameters* are declared by setting them in curly braces. Notice that no type signature is required for the implicit parameters in `_++_` and `map` as the types can be inferred from use. In contrast, a type signature must be supplied for the implicit parameters in the composition, `_∘_`.

```
module Whirlwind where ...
  infix 4 _≡_
    -- Propositional equality.
  data _≡_ {α : Set} (x : α) : α → Set where
    refl : x ≡ x

    -- Congruence under propositional equality.
  cong : ∀ {α β} (f : α → β) {x y} → x ≡ y → f x ≡ f y
  cong f refl = refl

    -- The map fusion theorem.
  thm-mapfusion : ∀ {α β γ} (f : β → γ) (g : α → β) →
                  ∀ xs → map f (map g xs) ≡ map (f ∘ g) xs
  thm-mapfusion f g []       = refl
  thm-mapfusion f g (x :: xs) = cong (_::_ (f (g x)))
                                     (thm-mapfusion f g xs)
```

Figure 7.2: Propositional equality and proofs on Lists.

7.2.2 Simple proofs

Thus far, Agda provides much the same functionality as Haskell 2010. Figure 7.2 introduces some of Agda's power beyond that of Haskell.

The data type _≡_ is known as *propositonal equality*. We may read it as follows. There exists a data type x ≡ y, where x and y both belong to an implicitly bound data type α. The only construction within this data type is refl, which is the definition of reflexivity. That is, y must be exactly x.

From propositional equality, we can derive the congruence lemma, cong. Reading the type signature from left-to-right, for all data types α and β and functions f from α to β, if x is equal to y, then f x is equal to f y.

Using the congruence lemma, we can prove the *map fusion* theorem (Wadler, 1981). This theorem states that mapping function f over the result of mapping function g over a list is equivalent to mapping the composition, f ∘ g, over the same list.

The proof is by induction over the list. In the base case, the results of both sides of the equality evaluate to [], so the proof is by reflexivity of equality. In the inductive case, the equality normalises to:

$$f\ (g\ x) :: map\ f\ (map\ g\ xs) \equiv f\ (g\ x) :: map\ (f \circ g)\ xs$$

The congruence lemma can meet this proof obligation. We apply the function _::_ (f (g x)) to both sides of the result of inductively/recursively applying the proof/function to the tail of the list.

```
module Whirlwind where ...
  open import Coinduction using (∞; ♯_; ♭)

    -- The CoNaturals.
  data CoNat : Set where
    zero :                    CoNat
    succ : (n : ∞ CoNat) → CoNat

    -- The lowest transfinite ordinal number in the CoNaturals.
  ω : CoNat
  ω = succ (♯ ω)

    -- CoNatural equivalence relation.
  data _≈_ : CoNat → CoNat → Set where
    zero :                                          zero   ≈ zero
    succ : ∀ {m n} → (m≈n : ∞ (♭ m ≈ ♭ n)) → succ m ≈ succ n

  thm-onemore : ω ≈ succ (♯ ω)
  thm-onemore = succ (♯ lem-n≈n ω)
    where
      lem-n≈n : ∀ n → n ≈ n
      lem-n≈n zero     = zero
      lem-n≈n (succ n) = succ (♯ (lem-n≈n (♭ n)))
```

Figure 7.3: Proofs on coinductive data types.

Proofs are possible in Agda because it appeals to the *Curry-Howard correspondence* (Curry and Feys, 1958; Howard, 1980). In a constructive logic, propositions can be considered as types and proofs can be considered values occupying those proposition-types. Inductive proofs over inductive data types become recursive proofs. Theorem proving is just functional programming!

7.2.3 Coinduction and guarded corecursion

Totality of functions is necessary for these proofs to be sound. Agda therefore requires all functions that perform case-distinctions to include clauses for all possible constructions. All recursive functions descend explicitly to a construction argument. A mechanism known as *coinduction* is used to express functions that do not naturally fit these constraints. Example definitions for describing a coinductive data type can be found in Figure 7.3.

We import several primitives from Agda's standard library. Note that, unlike Haskell, imports may be declared at any point of a module, not just the beginning.

One can view the ∞ primitive as a special purpose data type where $\infty\ \alpha$ has only one construction, $\sharp\ x$, where x is of type α. However, the effect of $\sharp$ is that

it introduces *laziness* at this point of the data structure. It leaves the argument unevaluated and unnormalised until it is forced by the ♭ primitive. Using ∞, ♯ and ♭ we can represent *infinite data types* in Agda and use them to describe proofs about infinite computations.

For example, consider the data structure CoNat, described in Figure 7.3. It is an implementation of the series of natural numbers. The argument of the successor function is *non-strict*. We can therefore implement a transfinite number, such as ω, because the recursive call will *not be normalised* unless a ♭ is applied.

We can perform proofs on this structure. The equivalence relation _≈_ is satisfied by conatural numbers that are the same. The succ construction of the relation is interesting for two reasons. *(1)* The m≈n argument is itself coinductive and *(2)* it shares the name of the second construction of CoNat. Agda permits constructions in different data types to share names as the intended construction can be inferred from the type context.

Using the _≈_ data type, we can prove the theorem thm-onemore that the successor to ω is an equivalent number. If we peel off the first succ of each side of the equivalence, then what remains is $\omega \approx \omega$. This can be proved with a lemma demonstrating the reflexivity of the conatural equivalence relation.

7.2.4 Logical constructs

So far, we have seen universal quantification but few other logical constructs. In this subsection and Figure 7.4, we introduce a few more.

Existentials in Agda are represented through a *dependent product*. We implement this using an Agda *record type*, a data type with precisely one construction. We may read the declaration of ∃ as follows. Given some data type α and predicate on α values, we must be able to construct some value of α and prove the predicate for that value.

Shorthand functions are provided for constructing — _, _ — and accessing the arguments of — proj_1 and proj_2 — the values of this data type. This dependent product type is also a generalisation of a traditional product type, as shown by the definition of _ × _. Record types can also be viewed as a module and we can expose the field accessors using the **open** keyword.

In Agda, we represent *falseness* through absurdity. The data type ⊥ has *no constructions*. Negation of a proposition, ¬, is the implication that the ability to construct it would result in the unconstructable absurdity.

A data type Dec represents decidable propositions. A yes-proof shows directly that p belongs to the set P of true propositions. A no-proof shows that if p were constructable, it would be absurd.

```
module Whirlwind where ...
  infixr 4 _,_
    -- Existential quantification.
  record ∃ {α : Set} (P : α → Set) : Set where
    constructor _,_
    field
      proj₁ : α
      proj₂ : P proj₁

  open ∃

  infixr 2 _×_
    -- Pair or logical cojunction.
  _×_ : Set → Set → Set
  α × β = ∃ {α} (λ _ → β)

    -- Empty or "absurd" set, representing false.
  data ⊥ : Set where
      -- No constructions for ⊥

    -- Negation is absurdity.
  ¬ : Set → Set
  ¬ P = P → ⊥

    -- Decidability of a proposition.
  data Dec (P : Set) : Set where
    yes : (p : P)     → Dec P
    no  : (¬p : ¬ P) → Dec P
```

Figure 7.4: Some logical constructions in Agda.

```
module Whirlwind where ...
  module WithSetoid where
    postulate
      -- Assume some set.
      α       : Set
      -- Assume there is decidable equaility for that set.
      _≡?_ : (x y : α) → Dec (x ≡ y)

      -- Proof of the injectivity of _::_.
    lem-consinj : ∀ {α} {x y : α} {xs} {ys} →
                  (x :: xs ≡ y :: ys) → (x ≡ y × xs ≡ ys)
    lem-consinj refl = (refl , refl)

      -- Tests if a list is a prefix of another. Correct-by-construction.
    isPrefix : (xs zs : List α) → Dec (∃ λ ys → xs ++ ys ≡ zs)
    isPrefix []       zs       = yes (zs , refl)
    isPrefix (x :: xs) []      = no (λ {(ys , ())})
    isPrefix (x :: xs) (z :: zs) with x ≡? z | isPrefix xs zs
    isPrefix (x :: xs) (.x :: zs) | yes refl | yes (ys , sound)
      = yes (ys , cong (_::_ x) sound)
    isPrefix (x :: xs) (z :: zs) | no x≢z | _
      = no (λ {(ys , sound) → x≢z (proj₁ (lem-consinj sound))})
    isPrefix (x :: xs) (z :: zs) | _ | no ¬sound
      = no (λ {(ys , sound) → ¬sound (ys , proj₂ (lem-consinj sound))})
```

Figure 7.5: A correct-by-construction implementation of isPrefix.

7.2.5 Correctness-by-construction

Figure 7.5 uses these logical constructs define a *correct-by-construction* implementation of a list prefix test. This principle of correctness-by-construction through the inhabiting of *properties-as-types* is how we intend to approach verifying the abstract supercompiler.

First, we declare a nested module called WithSetoid. Within its scope, we assume, or **postulate**, the existence of a data type α and a decidable equivalence function over it. Once we prove the trivial lemma that the _::_ construction is injective, we can define the isPrefix function.

Reading the type signature, *isPrefix* takes two lists xs and zs and *decides* if it is the case that there *exists* another list ys such that if we were to concatenate it to xs, we would get zs.

We define the correct-by-construction function *isPrefix* as follows:

- First we consider the case where xs is empty. An empty list is the prefix of any list and therefore the proposition holds, trivially. The suffix required

by the existential is precisely zs and both sizes of the equality normalise to zs so the proof is reflexivity.

- The second case is where xs is non-empty but zs is empty. There is nothing that we could concatenate onto a non-empty list to make it empty. Therefore, the decision is no with the proof that any value of ys immediately results in unconstructable absurdity.
- When xs and zs are both non-empty, we decide if their heads are equivalent. We also recursively call the isPrefix correct-by-construction function on the tails.
  - If the heads are equivalent and the tails form a prefix then this is a prefix. The suffix is ys, the suffix for the tails. We extend the proof using the congruence lemma from Section 7.2.2. Notice that the head of both lists is now x as a result of the equivalence proof-construction.
  - If the heads are not equivalent, then the function decided no. We show that for any suffix ys, the heads would have to be equivalent for the *isPrefix* correctness property to hold. This results in absurdity through contradiction.
  - If isPrefix on the tails of xs and zs decides no, then isPrefix xs zs must also decide no. We show that for any suffix ys, if *isPrefix* $(x :: xs)$ $(z :: zs)$ were to decide *yes*, then the *isPrefix xs zs* must necessarily decide *yes*. This results in absurdity through contradiction.

Thus, *isPrefix xs zs* not only decides whether a list *xs* is a prefix of *zs* but also *(a)* if *xs* is a prefix of *zs*, returns a suffix *ys* and a proof that $xs +\!\!+ ys \equiv zs$ and *(b)* if *xs* is not a prefix of *zs*, returns a proof that if there existed a *ys* such that $xs +\!\!+ ys \equiv zs$, then absurdity through a contradiction would occur. These properties are all expressed by the type signature of the *isPrefix* function and checked by the type-checker. The definition is correct-by-construction.

```
open import Data.Sum using (_⊎_)

postulate
  Syntax    : Set
  State     : Set
  initState : Syntax → State
  Value     : Set
  step      : State  → Value ⊎ State
```

Figure 7.6: Abstract definitions for language syntax and semantics.

```
open import Category.Monad.Partiality using (_⊥; now; later)
open import Coinduction using (♯_)
open import Data.Sum using (inj₁; inj₂)

run : State → Value ⊥
run s with step s
... | inj₁ v  = now v
... | inj₂ s' = later (♯ run s')
```

Figure 7.7: Small step to big step semantics.

7.3 SOURCE LANGUAGE AND EVALUABLE TYPES

A supercompiler consumes and produces abstract syntax trees. In our abstract model, let us assume (or **postulate**) the existence of some data type representing abstract syntax trees, as in Figure 7.6.

The programming language has an operational semantics represented by the step function. The abstract machine operates on intermediate states and each operational step will either return another step or a final value.

In Figure 7.7, we corecursively define a function that fully evaluates an abstract machine state. The *partiality monad* (Danielsson, 2012) encapsulates functions that take an indeterminate number of reductions to complete. One benefit of this construction is that it can be used to compare the number of reductions between two different functions, as we shall discuss in Section 7.4.

The Syntax and State data types can both be evaluated to a Value. We define a set of *evaluable* types in Figure 7.8, representing each of them as a record type. The first field, defines the type in question. The second, eval, holds the canonical evaluation function for this type. A Value is already evaluated so the evaluation function just states that the result is available immediately. A State is evaluated using the run function. Syntax trees are transformed into initial states before they are run.

```
open import Function using (_∘_)

record Evaluable : Set₁ where
  constructor ⟨_|_⟩ₑ
  field
    typeₑ : Set
    evalₑ : typeₑ → Value ⊥

open Evaluable

Valueₑ : Evaluable
Valueₑ = ⟨ Value | now ⟩ₑ

Stateₑ : Evaluable
Stateₑ = ⟨ State | run ⟩ₑ

Syntaxₑ : Evaluable
Syntaxₑ = ⟨ Syntax | run ∘ initState ⟩ₑ
```

Figure 7.8: The definition of evaluable types.

7.4 VERIFIED FUNCTIONS

The correctness of a supercompiler is defined by a relationship between the results of evaluating the input programs and the output programs. This relationship could be an equivalence such as *"input and output must evaluate to the same value"* or an ordering such as *"output must evaluate with no more reductions than the input."* Taking a minimal definition, let us assume some *refinement* relation (Back, 1988; Dijkstra, 1972) between evaluations to Value types. The refinement must be a preordering — reflexive and transitive. We postulate its existence in Figure 7.9.

Given this correctness relation, we define *verified functions* over evaluable types for which the relation holds. These are represented by the $\_\curvearrowright\_$ record type, parametrised by Evaluable types representing function inputs and outputs. The first field holds the plain Agda function. The second holds the proof that this function's inputs and outputs are related. The full-stop prefix for the second field marks it as *computationally irrelevant* — the value of this field is not used beyond the type-checker. It is useful for reducing type-checking complexity and could potentially simplify code extraction (Cruz-Filipe and Spitters, 2003).

In Figure 7.10 we define the verified functional identity and composition, exploiting the reflexive and transitive properties of the correctness relation.

It will be useful to extend our notion of evaluable data types to include *sums of evaluable types*, as in Figure 7.11. A selection function, $[\_,\_]_V$, can be used to choose a verified function based on the input sum.

```
postulate
    -- Refinement partial ordering on partial values.
  _◁_    : Value ⊥ → Value ⊥ → Set
    -- Refinement is reflexive.
  ◁-refl   : ∀ x → x ◁ x
    -- Refinment is transitive.
  ◁-trans : ∀ {x y z} (x◁y : x ◁ y) (y◁z : y ◁ z) → x ◁ z

  -- Verified functions, those that preserve refinement of values.
infix 0 _↷_
record _↷_ (α β : Evaluable) : Set where
  constructor _IsVerifiedBy_

  field
    func_v  : type_e α → type_e β
    .◁prf_v : ∀ (x : type_e α) → eval_e α x ◁ eval_e β (func_v x)
```

Figure 7.9: Refinement relation and verified functions.

```
id : ∀ {α : Set} → α → α
id x = x

id_v : ∀ {α} → α ↷ α
id_v {α} = id IsVerifiedBy (◁-refl ∘ eval_e α)

infixr 10 _⨟_ _⨟_v_
_⨟_ : ∀ {α β γ : Set} → (α → β) → (β → γ) → (α → γ)
f ⨟ g = g ∘ f

_⨟_v_ : ∀ {α β γ : Evaluable} → (α ↷ β) → (β ↷ γ) → (α ↷ γ)
(f IsVerifiedBy p) ⨟v (g IsVerifiedBy q)
  = (f ⨟ g) IsVerifiedBy (λ x → ◁-trans (p x) (q _))
```

Figure 7.10: Identity and composition for verified functions.

```
open import Data.Sum using ([_,_]; [_,_]′)

infixr 1 _⊎e_
  -- Sums of evaluable types.
_⊎e_ : Evaluable → Evaluable → Evaluable
⟨ α | evalα ⟩e ⊎e ⟨ β | evalβ ⟩e = ⟨ α ⊎ β | [ evalα , evalβ ]′ ⟩e

  -- Selecting an verified function by input sum.
[_,_]v : ∀ {α β γ} → (α ↷ γ) → (β ↷ γ) → ((α ⊎e β) ↷ γ)
[ f IsVerifiedBy p , g IsVerifiedBy q ]v = [ f , g ]′ IsVerifiedBy [ p , q ]
```

Figure 7.11: Evaluable sums of evaluable types.

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl)

  -- Verified functions, proved by simple reflexivity.
trivialᵥ : ∀ {α β : Evaluable} → (f : typeₑ α → typeₑ β)
         → (prf : (evalₑ β ∘ f) ≡ evalₑ α) → α ↷ β
trivialᵥ {⟨ _ | ∘ (eval ∘ f) ⟩ₑ} {⟨ _ | eval ⟩ₑ} f refl
  = f IsVerifiedBy (◁-refl ∘ eval ∘ f)

exposeᵥ : ∀ {α β : Evaluable} → (typeₑ α → α ↷ β) → α ↷ β
exposeᵥ f = (λ x → funcᵥ (f x) x) IsVerifiedBy (λ x → ◁prfᵥ (f x) x)
```

Figure 7.12: Further verified function combinators.

Several other functions may be written to aid the definition of verified functions. The $trivial_v$ function, defined in Figure 7.12, uses a simple proof of reflexivity to show that a function respects the required condition. The $expose_v$ function uses the verified function input to construct or select a verified function to operate on that input. In effect, it shows that we can determine control flow on the input without breaking the refinement relation. These *verified function combinators* are used to construct the abstract supercompiler.

7.5 THE ABSTRACT SUPERCOMPILER

In Chapter 5, the supercompiler was decomposed into four components: *(1)* A *normaliser* that reduces terms. *(2)* A *terminator* that ensures normalisation stops. *(3)* A *splitter* that produces smaller terms for further supercompilation. *(4)* A *memoiser* that reuses the results of supercompiling equivalent terms.

The abstract supercompiler here follows the same model. It implements those operations for which appropriate information is available and exposes verified interfaces where definitions are omitted.

7.5.1 Normalisation

Normalisation (Section 5.3.1) is the process of applying the step function until a *normal form* is reached. Therefore, to write the verified simplification function, the step function must be shown to preserve the relation.

In Figure 7.13, the *simulation preorder relation* over values wrapped in the *partiality monad* is imported from Agda's standard library. We require an additional assumption, postulated as ◁-Respects-$\gtrsim$, that if the simulation relation holds, it must be the case that the refinement relation holds.

Given these properties, we can construct the verified step function, $step_v$. The proof, *prfStep* is as follows. For any state s:

```
open import Category.Monad.Partiality
  using (module Equality; module Equivalence)

open Equality {A = Value} (_≡_) using (_≳_; now; later′)
module ≳ = Equivalence

postulate
  ◁-Respects-≳ : ∀ {x y} → x ≳ y → x ◁ y

step_v : State_e ↷ Value_e ⊎_e State_e
step_v = step IsVerifiedBy (◁-Respects-≳ ∘ prfStep)
  where
    prfStep : ∀ s → run s ≳ [ now , run ]′ (step s)
    prfStep s with step s
    ... | inj₁ v  = now refl
    ... | inj₂ s' = later′ (≳.refl refl)
```

Figure 7.13: A verified version of the step function.

```
open import Data.Bool using (Bool; if_then_else_)

postulate
  isNormal : State → Bool

abstract
  {-# NO_TERMINATION_CHECK #-}
  normalise_v : State_e ↷ Value_e ⊎_e State_e
  normalise_v = step_v ⨟_v
                  [ trivial_v inj₁ refl
                  , expose_v (λ x → if isNormal x then trivial_v inj₂ refl
                                                  else normalise_v) ]_v
```

Figure 7.14: An abstract normaliser.

- If the next step, *step s*, results in it halting with value *v*, *run s* would return the same value.

- If the next step, *step s*, results in another intermediate state, *s′*, *run s* would return the same value as *run s′* allowing for one additional reduction.

In Figure 7.14 we assume the existence of a function, *isNormal*, that decides if a state is in normal form. Using *isNormal*, the verified step function, $step_v$, and the combinators introduced in Section 7.4, we can define $normalise_v$ — the verified normalisation function. No additional verification work is required as it is solely constructed from other verified functions.

```
open import Data.Maybe using (Maybe;maybe)

postulate
  TermHistory   : Set
  canContinue   : TermHistory → State → Maybe TermHistory
  emptyHistory  : TermHistory
```

Figure 7.15: An abstract terminator.

7.5.2 Termination

The termination of the supercompilation algorithm is *not* being verified. Therefore, the Agda termination checker must be disabled for the $normalise_v$ function (defined in Figure 7.14) using the `{-# NO_TERMINATION_CHECK #-}` pragma. The normalisation function may, indeed, not terminate depending on the definition of *isNormal*. The function must be marked as **abstract** to prevent possible non-termination of the Agda type-checker.

One must always be careful when disabling Agda's termination checker. As discussed in Section 7.2.3, totality is essential for proofs to be sound. A function that never produces a value satisfies all property-types. However, for the purposes of our model, it easier to prove outside of the Agda logic that *isNormal* will eventually return true. Similarly, proofs outside of Agda will be necessary for Figure 7.18.

Despite not including termination proofs in our abstract model, we must still define the conditions under which *driving* (Section 5.3.5) terminates and *splitting* begins. As discussed in Section 5.3.2, the choice of termination function differs widely in supercompilation implementations. It is very dependent on the choice of abstract syntax tree representation. However, as illustrated in Figure 7.15, the key concepts are quite abstract. Given some record of the supercompilation history, the termination component checks to see if the history can be extended with a given state. If it can, it returns this new history. If not, it is a signal that the termination condition has been triggered and normalisation must stop. The empty history is assumed as the initial value for the algorithm.

The property that the termination condition *must be* triggered in a finite number of steps is absent. A potential encoding would be that states form a well-quasi-ordering under the termination condition. However, it is not clear how to convey this information to Agda's internal termination representation. This issue will be discussed as further work in Section 9.5.

```
postulate
  BindingIn      : Set
  emptyBinding   : BindingIn
  BindingOut     : Set

record BindingCtx (α : Set) : Set where
  field
    ctx   : BindingOut
    value : α

postulate
  runBinding     : BindingCtx Syntax → State

BoundSyntax_e : Evaluable
BoundSyntax_e = ⟨ BindingCtx Syntax | run ∘ runBinding ⟩_e

Phase = BindingIn → State_e ↷ BoundSyntax_e

postulate
  finalBinding_v : BoundSyntax_e ↷ Syntax_e
  memoise_v      : Phase → Phase
  residualise_v  : Phase → Phase
```

Figure 7.16: An abstract memoiser.

7.5.3 Memoisation

The supercompiler must store and reuse results, not only for execution time performance but also to ensure that the supercompiler terminates for recursive languages. This requirement for memoisation was discussed in detail in Section 5.3.3.

Definitions for memoisation in the abstract supercompiler can be found in Figure 7.16. A *binding context* holds information about shared supercompilation results. The *runBinding* function takes partial syntax tree wrapped in a binding context and produces a complete syntax tree. An expression contained in a binding context can also be viewed as an evaluable data type.

The $memoise_v$ higher-order function will either *(a)* detect that the input state has been processed before and return a reference to that result in the binding context, or *(b)* process the state with the functional argument. Another function, $residualise_v$, extends the *BindingIn* binding information with a fresh binder for a new chain of normalisations.

```
postulate
  Split     : Set
  runSplit  : Split → State

Split_e : Evaluable
Split_e = ⟨ Split | run ∘ runSplit ⟩_e

postulate
  split_v            : State_e ↷ Split_e
  applySubterms_v    : (State_e ↷ BoundSyntax_e) → (Split_e ↷ BoundSyntax_e)
```

Figure 7.17: An abstract splitter.

7·5·4 Splitting

The final component of our abstract supercompiler splits a normal-form term into a number of smaller terms so that supercompilation can continue. In the reference implementation given in Section 5.3.3, the *split* function produced a number of smaller states and a method of recombining the results of their supercompilation into an expression.

Figure 7.17 defines a data structure *Split* for the results of splitting. Its precise definition construction is left abstract. Instead, the $split_v$ verified function splits the normal-form state as a *Split* value and the *applySubterms* combinator applies a verified supercompiler function to each subterm in a split before recombining the results with the context.

The *runSplit* function is used to define $Split_e$, the type for *Split*. It provides a semantics for the split data type by converting them back into states. See the instantiation of this signature in Section 8.5 for more detail.

7·5·5 Overall control algorithm

The definitions in Figure 7.18 complete the abstract model, combining the components to achieve the supercompiler process. We assume $finalState_v$, a verified function that describes how to turn semantic values into equivalent abstract machine states.

The rest of the control algorithm essentially follows Section 5.3.5. States are normalised in the *drive* phase until a termination condition is reached. They are then split into further states in the *tie* phase. Results are memoised to preserve sharing and global termination. Once again, the Agda termination checker is overridden because there is not encoding of a proof that this algorithm must terminate that satisfies Agda's termination checker.

```
postulate
  finalState_v : Value_e ↷ State_e

{-# NO_TERMINATION_CHECK #-}
mutual
  drive : TermHistory → Phase
  drive hist = memoise_v (drive' hist)

  drive' : TermHistory → Phase
  drive' hist binfo = normalise_v ⨟_v ([ (finalState_v ⨟_v tie hist binfo) ,
    expose_v (λ s → maybe drive (tie hist) (canContinue hist s) binfo) ]_v)

  tie : TermHistory → Phase
  tie hist binfo = split_v ⨟_v applySubterms_v (residualise_v (drive hist) binfo)

supercompile : Syntax_e ↷ Syntax_e
supercompile = trivial_v initState refl ⨟_v
               residualise_v (drive emptyHistory) emptyBinding ⨟_v
               finalBinding_v
```

Figure 7.18: An abstract supercompiler.

7.6 CODE EXTRACTION

It is possible to extract a Haskell program from an Agda source using the *MAlonzo* compiler. However, as discussed by Fredriksson and Gustafsson (2011), it is not a perfect solution.

> *"Agda's type system is more expressible [sic.] than Haskell's. The backend has to insert coercions around the result of all function calls and all function arguments to make the program pass the type checker."* (Fredriksson and Gustafsson, 2011)

This need for frequent explicit coercions impacts performance as many Haskell compiler optimisations are dependent on type information.

Consider the example of a character counting program. Figure 7.19 contains a naive Haskell solution and an Agda equivalent. The programs are very similar, discounting some additional Agda functions to perform type-conversions between Agda lists, strings and costrings.

However, the Haskell output from compiling the Agda variant looks nothing like the program consisting of 70 characters in Figure 7.19. The extracted Haskell contains 4,473 characters, mostly type-coercions, and this excludes all the library functions that are compiled into separate Haskell files. Executing the Haskell solution (compiled by GHC) and the Agda solution (compiled into Haskell and then by GHC) illustrates even more startling differences. The Haskell solution counts over 35 million characters per second. The Agda

```
  -- Word-count in Haskell
main :: IO ()
main = readFile "file.txt" >>= putStrLn ∘ show ∘ length

  -- Word-count in Agda
main : IO Unit
main = readFiniteFile "file.txt" >>=
         putStrLn ∘ toCostring ∘ show ∘ length ∘ toList
```

Figure 7.19: Character count program source in Haskell and Agda.

solution counts around 150 characters per second with files larger than a million characters causing a stack overflow. Fredriksson and Gustafsson (2011) carefully examine several other examples in their masters' thesis.

We conclude that using the MAlonzo compiler is not a reasonable method of producing an *"effective"* (using the Leroy sense of the word) verified supercompiler. However, the simplicity of our model does lend itself to being translated by-hand into a Haskell implementation, as we shall do in the next chapter.

7.7 SUMMARY

We have developed a more abstract model of a supercompiler than the Chapter 5 reference implementation. It is encoded in Agda, a dependently-typed language, so that we can incorporate correctness properties as types. The satisfaction of these properties is mechanically verified by Agda's type-checker.

The key areas of abstraction in this model over are; *(1)* the source language *(2)* the refinement preordering and *(3)* the termination condition. However, once an operational semantics and the *isNormal* function have been instantiated, the *normaliser* component of the supercompiler is not only complete but also verified for any definition of refinement!

The splitting and memoisation components of the supercompiler must be instantiated separately. However, the key result of the proof in this chapter is that they may also be verified independently, at the component-level. Through *verified functions*, we express at the type-level that correctness properties for particular functions must be fulfilled.

In Chapter 8, we shall reuse verified functions to integrate proof results from this abstract model and further verification results from property-based testing.

8 HYBRID VERIFICATION OF A SUPERCOMPILER

This chapter combines our previous contributions to verify a working supercompiler implementation. Using the verified abstract supercompiler from Chapter 7 as a foundation, we expose verifiable interfaces to a Haskell implementation and instantiate it with components from the supercompiler described in Chapter 5. We test the correctness of the unproven components using the techniques from Chapters 3 and 4.

8.1 INTRODUCTION

An abstract framework for defining verified supercompilers was introduced in Chapter 7. It showed how individual components of a supercompiler, discussed in Chapter 5, could be verified independently to give a proof of the entire system.

The normalisation and termination components were shown to be semantics preserving *by definition* for any instantiation of the framework. On the other hand, the definition of the memoisation and splitting components crucially depends on the particular source language.

Instantiating the abstract model with a specific language is difficult in Agda. The type-checking process slows considerably as more definitions are added. Even the extracted Haskell programs do not perform as well as equivalent programs in other functional languages, as we discussed in Section 7.6. Furthermore, verifying the final components using Agda-style proof is laborious and would be difficult to maintain if changes are made to the language and supercompiler implementation.

This chapter proposes the use of a *hybrid verification model* that combines the existing verification results from Chapter 7 with the property-based testing techniques discussed in Chapters 2 to 4. Our objective is to construct a supercompiler with an associated *verification program* that tests the correctness of unproven components. The verification program's output may take the form:

```
 $ ./Mock-Chapter8
* Assumed correctness of component_v1
* Semantic preservation over component_v2
Passed after n2 tests.
* Semantic preservation over component_v3
Failed with counterexample:
...
```

Hybrid verification is achieved by translating the abstract Agda model into a Haskell definition. In this way we benefit from lower compilation and execution times from a more mature language implementation, GHC. Then we use the new version of the Lazy SmallCheck library (described in Chapter 3) to test properties and assumptions that are not already covered by the Agda proof. In general, inline code listings are used for type-class specifications while code listings in figures are used for implementation detail.

The key questions answered by this chapter are:

- How can the features of Haskell be exploited to represent missing Agda functionality? *(Sections 8.2 and 8.3)*
- How can the Chapter 7 definition of verified functions be translated to use property-based testing instead of properties-as-types? *(Section 8.2)*
- How can the canonical test programs generated in Chapter 4 be used to produce test-data for our correctness properties? *(Section 8.4)*
- What changes, if any, are required to the supercompiler in Chapter 5 to fit this hybrid verification model? *(Section 8.5)*
- What are the short-comings of our hybrid verification model and what improvements may be made? *(Sections 8.5 and 8.6)*

Section 8.2 shows how evaluable types and verified functions are represented in the Haskell implementation. Section 8.3 translates the Agda abstract supercompiler into Haskell. Section 8.4 generically creates valid syntax trees, states, splits and final values. Section 8.5 gives an instantiation of the abstract supercompiler for our source functional language. Section 8.6 records the results and lessons learned from applying our hybrid verification to the supercompiler.

8.2 HYBRID VERIFICATION MODEL

In Chapter 7, we introduced the concepts of *verified functions* over *evaluable types*. Signatures comprised of these type-constructions were presented as interfaces to our abstract implementation. What follows is a translation of these concepts to Haskell and a property-based testing verification paradigm. In general, Haskell type-classes take the place of Agda postulates. Lazy SmallCheck properties take the place of Agda properties-as-types.

8.2.1 Evaluable types

An *evaluable type* (Section 7.3) has an associated function for evaluating values to some result. The expression of evaluable types in Haskell is very simple if we

use a widely adopted language extension — *type families* (Schrijvers et al., 2008). An instance *Evaluable a* defines both a result type $Result_e\ a$ and an evaluation function $eval_e$ that takes values of type a to $Result_e\ a$.

```
class Evaluable a where
  type Result_e a
  eval_e :: a → Result_e a
```

8.2.2 Verified functions

Verified functions, on the other hand, require a different representation in Haskell compared to Agda. Verified functions over evaluable types (Section 7.4) maintain some semantic relationship between their inputs and outputs. Haskell's type specifications cannot depend on values and, therefore, cannot represent the $\lhd prf$ proof obligation. Instead, we shall use property-based testing in place of the constructive proofs.

In the Agda model, the proof obligations are marked as *computationally irrelevant*. This means that the proof values may be erased at runtime with no loss of *computational behaviour*. We will trust any function verified in the Agda model to satisfy the semantic correctness property when translated into Haskell. The two language share very similar semantics and without a more formal and trusted translation mechanism, such as the compiler described in Section 7.6, it is necessary for us to use this assumption.

This leaves the case where a component requires further implementation details. If we intend to verify these components using property-based testing, we must consider three areas where it differs from mechanical proof:

1. Agda reports invalid proof-values by identifying them through their property-types and line number in the source code. In Haskell property-based testing, we must label our properties in such a way that they can be identified when falsified.

2. A constructive proof may rely on recursion to share common proof-values that are applied recursively/inductively. If we are not careful, using the same approach will lead to an infinite set of properties to test.

3. In our Agda model in Chapter 7, not all function arguments are relevant to the semantics preservation properties. These are passed in as non-verified functional arguments, such as the *TermHistory* argument in the definition of *drive* in Section 7.5.5. We must incorporate these arguments into the Haskell implementation of verified functions so that they can be instantiated with test-data.

```
type Assumptions = Map AsmDesc (Maybe Property)
type AsmDesc = String
data Verified r a b where
  IsVerified :: (r → a → b) → Assumptions → Verified r a b
  Comp       :: Verified r b c → Verified r a b → Verified r a c
  Merge      :: Verified r a b → Verified r a' b → Verified r (Either a a') b
  Either     :: Verified r a b → Verified r' a b → Verified (Either r r') a b
  Modify     :: (r → a → r') → Verified r' a b → Verified r a b
type r ⇝ a ↷ b = Verified r a b
type a ↷ b = Verified () a b
```

Figure 8.1: Verified functions in Haskell

Figure 8.1 shows our implementation of verified functions in Haskell taking into account these concerns. Obligations, or *Assumptions*, are represented as a map of descriptions to Lazy SmallCheck properties. The assumption descriptions are labels to identify any falsified properties for the user.

The type *Verified* is ternary as opposed to the binary $(\curvearrowright)$ in Chapter 7. These allow for the functional arguments that do not impact the semantic preservation properties. The unit type, $()$, may be used when verified functions do not require an auxiliary argument. We provide a shorthand type synonym, $(\curvearrowright)$, for these occasions.

Our Haskell implementation reifies a subset of the primitive operations on verified functions. This *deep embedding* allows us to use the technique described by Gill (2009) to recover sharing and so detect recursive definitions — see Section 8.2.4. The internals of a *Verified* data structure should, as far as possible, be hidden from the framework user. The following subsections describe the combinators for building verified functions such that properties are maintained.

8.2.3 Construction of verified functions

Two alternative methods for lifting Haskell functions into verified functions are shown in Figure 8.2. If the function has already been shown to preserve semantics through Agda proof, the *trustMe* combinator may be used. Alternatively, *testMe* pairs the pure function with the runtime property that it preserves the semantics of the input value. As an equivalence relation, it satisfies the refinement pre-ordering condition from Section 7.4. These two combinators reflect the essence of the hybrid verification methodology, incorporating verification attained through theorem proof and property-based testing respectively.

```
  -- Trust that a function preserves semantics.
trustMe :: String → (r → a → b) → (r ⇝ a ↷ b)
trustMe str f = IsVerified f (Map.singleton str' Nothing)
  where str' = "Assumed correctness of " ++ str

  -- Test that a function preserves semantics.
testMe :: (Evaluable a, Evaluable b, Serial r, Serial a, Eq (Result_e a)
        , Result_e a ~ Result_e b) ⇒ String → (r → a → b) → (r ⇝ a ↷ b)
testMe str f = IsVerified f (Map.singleton str' (Just prop))
  where str' = "Semantic preservation over " ++ str
        prop = forAll $ λr x → eval_e x ≡ eval_e (f r x)
```

Figure 8.2: Converting Haskell functions into verified functions

The *trustMe* combinator just stores an identifier with no property so that it is reported but not checked when the program is verified. The *testMe* combinator requires that the types of the inputs and outputs to the Haskell function are *Evaluable* and the input types are *Serial*. The combinator constructs a Haskell verified function that carries the obligation that input and output evaluate to the same value. A slight modification will be made to *testMe* in Section 8.4 to cope with some peculiarities of the method used to generate test-data.

Figure 8.21 (on page 126 at the end of this chapter) describes a collection of combinators over verified functions. As with the Agda model, we can implement standard function-like combinators such as identity and composition. Verified functions are also defined for evaluable sums and conditionals, supported by proofs in the Agda model.

8.2.4 Extraction of implementations and obligations

A normal Haskell function can be extracted from a verified function by applying $func_v$, defined in Figure 8.3. It discards the assumptions of an *IsVerified* construction and replaces the other reification constructions with appropriate Haskell functions.

Figure 8.4 contains the definition of a function, *depthCheckAssumptions* that uses Lazy SmallCheck to test the assumptions of a verified function up to a defined depth. The *Data.Reify* library (Gill, 2009) is used to expose any recursion in our definitions and ensure that the set of properties to be checked can be computed in finite time. This is achieved by defining an instance of *MuRef* (*Verified r a b*) that maps our recursive data-structure to explicit graphs of nodes *ArrowNode n*, where *n* is the type of unique labels for each node. The function component of the *Verified* data-structure is discarded and only the *Assumptions* are retained.

```
func_v :: (r ⇝ a ↷ b) → r → a → b
func_v (IsVerified f _) r = f r
func_v (Comp     f g) r = func_v f r ∘ func_v g r
func_v (Merge    f g) r = either (func_v f r) (func_v g r)
func_v (Either   f g) r = either (func_v f) (func_v g) r
func_v (Modify m f)   r = func_v f =≪ m r
```

Figure 8.3: Extracting a Haskell function from a verified function.

```
  -- Labelled-graph model of verified functions.
data ArrowNode r = ArrowNode Assumptions [r]
  -- Recovery of sharing using Gill's (2009) technique.
instance MuRef (Verified r a b) where
  type DeRef (Verified r a b) = ArrowNode
  mapDeRef f (IsVerified _ ps)
    = pure $ ArrowNode ps []
  mapDeRef f (Comp     g h)
    = ArrowNode Map.empty <$> sequenceA [f g, f h]
  mapDeRef f (Merge    g h)
    = ArrowNode Map.empty <$> sequenceA [f g, f h]
  mapDeRef f (Either   g h)
    = ArrowNode Map.empty <$> sequenceA [f g, f h]
  mapDeRef f (Modify   _ g)
    = ArrowNode Map.empty <$> sequenceA [f g]
extractAssumptions :: (r ⇝ a ↷ b) → IO Assumptions
extractAssumptions f = do
  Graph gr _ ← reifyGraph f
  return $ Map.unions [ps | (_, ArrowNode ps _) ← gr]
  -- Display and check testable assumptions in a verified function.
depthCheckAssumptions :: Int → (r ⇝ a ↷ b) → IO ()
depthCheckAssumptions d f = do
  ps ← extractAssumptions f
  sequence_ [do putStrLn ("* " ++ str)
                when (isJust p) $ depthCheck d (fromJust p)
             | (str, p) ← Map.toList ps]
```

Figure 8.4: Testing assumptions of verified functions

8.3 AN ABSTRACT SUPERCOMPILER IN HASKELL

Now we can translate the abstract supercompiler from Chapter 7 into Haskell. Once again, we shall use Haskell's type-classes extended with type families in place of Agda's postulates.

8.3.1 Languages

A type-class instance, *Language* ℓ, defines the data types and functions needed for a language (labelled ℓ) with a small-step operational semantics. The following class deceleration is a direct translation of the Section 7.3 of the Agda abstract model.

```
class Language ℓ where
  data Syntax ℓ
  data Value ℓ
  data State ℓ
  initState :: Syntax ℓ → State ℓ
  step :: State ℓ → Either (Value ℓ) (State ℓ)
```

In Figure 8.5, we use instances of these signatures to derive a function *run*. We also define, for any *Language* ℓ, *Syntax* ℓ, *Value* ℓ and *State* ℓ instances for the *Evaluable* type-class. These definitions are the similar to the Agda model in Section 7.3. However, given that our comparisons will be based on runtime results, we need to ensure that our evaluations terminate, even when the test-program is non-terminating. In Figure 8.5, we use the *runFor* combinator to limit our evaluations to a finite number of operational steps.

Appealing to these definitions of *Evaluable* and the proofs in Figure 7.13, we claim that the *initState* and *step* functions preserve any correctness property, defining verified versions in Figure 8.6. These claims are expressed using the *trustMe* method of introducing verified functions.

8.3.2 Normalisation

Any instance of *Language* ℓ can be extended with an *isNormal* function to form an instance of *Normaliser* ℓ.

```
class Language ℓ ⇒ Normaliser ℓ where
  isNormal :: State ℓ → Bool
```

A generic, normalisation function $normalise_v$ is defined in Figure 8.7. It follows the definition given in Section 7.5.1 but has been adjusted to use the *cond_v* combinator from Figure 8.21.

```
  -- Run from a given state until termination
run :: Language ℓ ⇒ State ℓ → Value ℓ
run = either id run ∘ step

instance Language ℓ ⇒ Evaluable (Value ℓ) where
  type Result_e (Value ℓ) = Maybe (Value ℓ)
  eval_e = Just

  -- Run a state for a finite number of steps
runFor :: Language ℓ ⇒ Int → State ℓ → Maybe (Value ℓ)
runFor 0 s = Nothing
runFor n s = either Just (runFor (pred n)) $ step s

instance Language ℓ ⇒ Evaluable (State ℓ) where
  type Result_e (State ℓ) = Maybe (Value ℓ)
  eval_e = runFor 1000

instance Language ℓ ⇒ Evaluable (Syntax ℓ) where
  type Result_e (Syntax ℓ) = Maybe (Value ℓ)
  eval_e = eval_e ∘ initState
```

Figure 8.5: Definitions over *Language* instances.

```
initState_v :: Language ℓ ⇒ Syntax ℓ ↷ State ℓ
initState_v = trustMe "initState_v" $ const initState

step_v :: Language ℓ ⇒ State ℓ ↷ Either (Value ℓ) (State ℓ)
step_v = trustMe "step_v" $ const step
```

Figure 8.6: 'Trusted' verified forms of *initState* and *step*.

```
normalise_v :: Normaliser ℓ ⇒ State ℓ ↷ Either (Value ℓ) (State ℓ)
normalise_v = step_v >>> left_v ||| cond_v (const isNormal) right_v normalise_v
```

Figure 8.7: Normalisation for instances of *Normaliser*.

8.3.3 Termination

Any language instance can also be extended with information about a supercompilation termination condition (Section 5.3.2). As with the Agda abstract model in Section 7.5.2, an instance must define a data type containing historical information, an empty history and a method for determining if the termination condition has been triggered.

```
class Language ℓ ⇒ Terminator ℓ where
  data History ℓ
  emptyHistory :: History ℓ
  canContinue  :: History ℓ → State ℓ → Maybe (History ℓ)
```

8.3.4 Memoisation

Memoisation is required to reuse the results within a supercompilation and is essential for supercompiling recursive programs, as was discussed in Section 5.3.4. Another type-class determines the method by which a supercompiler instance memoises results. Each particular language instance provides data types to hold binding information, a verified function $memoise_v$ for memoising supercompiler phases and a verified function $residualise_v$ that indicates a point at which a new binder may be generated.

```
class Language ℓ ⇒ Memoiser ℓ where
  data BindingIn    ℓ
  data BindingOut ℓ
  memoise_v    :: Phase_v ℓ → Phase_v ℓ
  residualise_v :: Phase_v ℓ → Phase_v ℓ
  emptyIn      :: BindingIn ℓ
  runBWriter :: BoundSyntax ℓ → Syntax ℓ
data BWriter       ℓ a = BWriter (BindingOut ℓ) a
type BoundSyntax ℓ    = BWriter ℓ (Syntax ℓ)
type Phase_v ℓ = (BindingIn ℓ, History ℓ) ⇝ State ℓ ↷ BoundSyntax ℓ
```

The *runBWriter* function is used to retrieve syntax in a binding context. We use it to define an evaluable type for such syntax. As *runBWriter* is shown to be correct-by-construction in Section 7.5.3, we define a verified variant $runBWriter_v$ in Figure 8.8.

8.3.5 Splitting

Splitting is the process of taking a state in a normal form and separating it into supercompilable subterms, as was described in Section 5.3.3. An instance *Splitter* ℓ defines a *Split* ℓ data-structure that holds these subterms and a residual context, the method of combining the results of their supercompilation into a complete expression.

The verified function $split_v$ turns intermediate states into splits while the $applySubterms_v$ applies a verified function to each subterm in the split before recombining the results into an expression. The function *runSplit* is used to recover a state from a split. An *Evaluable* instance is given in Figure 8.9, using *runSplit* to define the semantics of a split.

```
class Language ℓ ⇒ Splitter ℓ where
  data Split ℓ
  split_v          :: State ℓ ↷ Split ℓ
  applySubterms_v  :: Phase_v ℓ →
                      (BindingIn ℓ, History ℓ) ⇝ Split ℓ ↷ BoundSyntax ℓ
  runSplit         :: Split ℓ  → State ℓ
```

8.3.6 Supercompiler

Finally a type-class *Supercompiler* ℓ depends on all the previous type-classes representing supercompiler components. In addition to these, it defines $finalState_v$ — a verified function for retrieving a state from a final value.

```
class (Normaliser ℓ, Terminator ℓ, Memoiser ℓ, Splitter ℓ) ⇒
      Supercompiler ℓ where
  finalState_v :: Value ℓ ↷ State ℓ
```

A supercompiler is defined for instances of the *Supercompiler* type-class in Figure 8.10. It follows the definitions in Section 7.5.5 but, once again, is adjusted for the ternary verified functions types.

```
instance Memoiser ℓ ⇒ Evaluable (BoundSyntax ℓ) where
  type Result_e (BoundSyntax ℓ) = Maybe (Value ℓ)
  eval_e = eval_e ∘ runBWriter
runBWriter_v :: Memoiser ℓ ⇒ BoundSyntax ℓ ↷ Syntax ℓ
runBWriter_v = trustMe "runBWriter_v" $ const runBWriter
```

Figure 8.8: Evaluable types and verified functions for binding contexts.

```
instance Splitter ℓ ⇒ Evaluable (Split ℓ) where
  type Result_e (Split ℓ) = Maybe (Value ℓ)
  eval_e = eval_e ∘ runSplit
```

Figure 8.9: Evaluable instances for splits.

```
supercompile :: Supercompiler ℓ ⇒ Syntax ℓ ↷ Syntax ℓ
supercompile = initState_v >>>
               apply_v drive (emptyIn, emptyHistory) >>>
               runBWriter_v
drive :: Supercompiler ℓ ⇒ Phase_v ℓ
drive = memoise_v drive'
drive' :: Supercompiler ℓ ⇒ Phase_v ℓ
drive' = const_v normalise_v >>> (const_v finalState_v >>> tie) |||
                                 (maybe_v canContinue' tie drive)
  where canContinue' (r, h) x = (, ) r <$> (canContinue h x)
tie :: Supercompiler ℓ ⇒ Phase_v ℓ
tie = const_v split_v >>> applySubterms_v (residualise_v drive)
```

Figure 8.10: A verified abstract supercompiler in Haskell.

```
  -- Only ASTs that are well-typed and canonical.
applyFilter :: ProgGen.ProR → Filtered (ProgGen.ProR)
applyFilter x = Filtered [ProgGen.good x] x

instance Serial (Filtered ProgGen.ProR) where
  series        = applyFilter <$>^ series
  seriesWithEnv = applyFilter <$>^ seriesWithEnv
```

Figure 8.11: Serial instance of filtered abstract syntax trees.

8.4 TEST DATA–VALUES

As discussed at length in Chapters 2 to 4, the method by which we generate test-data for our properties is crucial. In Chapter 4, we presented a method for generating canonical programs for a first-order functional language. We shall reuse this work as a basis for our test-data for supercompiler verification.

```
import qualified Chapter2 as ProgGen
```

Recall that the program-generation work in Chapter 4 was for a much smaller language than that used in Chapter 5. Also, in Chapter 4 only well-formed, canonical abstract trees were generated. As we have not yet produced a method for non-canonical program selection (see Section 9.3), we shall use this set of canonical representatives. We will also need to generate values, states, splits and binding contexts if we are to test all the properties generated by the hybrid verification model.

Recall that the approach take in Chapter 4 was to generate all possible abstract syntax trees and then apply lazy predicates as antecedents to filter away anything that is not well-formed, well-scoped, well-typed or canonical. The use of antecedent filtering conditions does not immediately fit with the formulation of the property associated with *testMe*, defined in Section 8.2.3. In our original formulation, the property expects to freely generate test-data with no constraints.

Therefore, to encapsulate this filtering behaviour at the type-level, we shall wrap our abstract syntax trees (and other evaluable types) in a *filtering context* which contains the value and properties deciding if it is valid.

```
data Filtered a = Filtered { filterPreCond :: [Property]
                           , filterResult  :: a }
```

We can now define the series of *filtered abstract syntax trees* for the core language described by Figure 4.5. In Figure 8.11, the *applyFilter* function adds

```
instance Functor Filtered where
  fmap f (Filtered ps x) = Filtered ps (f x)
instance Monad Filtered where
  return = Filtered []
  Filtered ps x >>= f = Filtered (ps ++ qs) fx
    where (Filtered qs fx) = f x
emptyFilter :: Filtered a
emptyFilter = Filtered [ff] $ error "emptyFilter: Bad precondition_v"
runFilter :: Filtered Bool → Property
runFilter (Filtered [] x) = mkProperty x
runFilter (Filtered ps x) = foldr1 (∧par) ps ⟹ x
```

Figure 8.12: Monadic construction and evaluation of filtered properties.

the *ProgGen.good* predicate to a syntax tree as a filter. *ProgGen.good* combines all the principles discussed in Chapter 4. As there is no obvious method for showing *Filtered* types, we override the *seriesWithEnv* default to display the wrapped *Core.ProR* value directly.

A monadic interface is presented to the *Filtered* data types, as defined in Figure 8.12. Using these, a *Filtered Property* can be constructed which can be converted into a *Property* using the *runFilter* combinator.

Test programs for a supercompiler can be created from the programs in the language from Figure 4.5. A type-class *Convertable* ℓ contains a function *convert* that translates programs in the *ProgGen.ProR* to the appropriate syntax representation.

```
class Language ℓ ⇒ Convertable ℓ where
  convert :: ProgGen.ProR → Syntax ℓ
```

Use this *convert* function, we can generically define filtered series for *Syntax* ℓ, *State* ℓ, *Value* ℓ and *Split* ℓ as shown in Figure 8.14. The supercompiler source syntax trees are created directly by *convert*. Intermediate states are generated by running a syntax tree for a number of operational semantic steps. We use *Peano* numerals to represent the number of steps so that *Lazy SmallCheck* can prune away any value for the number of steps that goes beyond halting or crashing.

Final values, on the other hand, are created from syntax trees that terminate with a value after being run for a finite number of steps. Splits, of the form described in Section 8.3.5, are created by applying a split operation to intermediate states.

```
testMe′ :: (Evaluable a, Evaluable b, Serial (Filtered r), Serial (Filtered a)
           , Eq (Result_e a), (Result_e a ∼ Result_e b)) ⇒
           String → (r → a → b) → (r ⇝ a ↷ b)
testMe′ str f = IsVerified f (Map.singleton str′ (Just prop))
  where str′ = "Semantic preservation over " ++ str
        prop = forAll $ λr x → runFilter $ do
          r′ ← r
          x′ ← x
          return $ eval_e x′ ≡ eval_e (f r′ x′)
```

Figure 8.13: Semantics preserving functions using *Filtered* series.

Generating termination histories and binding information depends crucially on the particular language instantiation. They cannot be defined generically but must be defined specifically for each language. See Section 8.6.

A new variant of the *testMe* combinator is defined in Figure 8.13. It quantifies over filtered values, using *runFiltered* and the monadic interface from Figure 8.12 to ensure that the filter preconditions are applied before the semantics preservation property is tested. The *testMe′* combinator variant will be used to introduce property-based tested functions in our supercompiler instantiation.

8.5 A VERIFIED IMPLEMENTATION

We shall now instantiate the type-classes in Section 8.3 with the components from the supercompiler defined in Chapter 5. We begin by importing our previous definitions and declaring a type-level tag for our core language.

```
import qualified Chapter5 as Core
data Core
```

For many parts of the core supercompiler, our instances are merely wrappers around the previously defined *Core* functions imported from Chapter 5. This is the case for the *Language Core*, *Normaliser Core* and *Terminator Core* instances in Figure 8.15 which take their definitions from Sections 5.2, 5.3.1 and 5.3.2 respectively.

MEMOISER We need to make modifications to the implementation of the memoiser component from Section 5.3.4. The *Memoiser Core* instance is defined in Figure 8.16, making use of the auxiliary functions defined in Figure 8.22

```
  -- Generating syntax trees
instance (Convertable ℓ) ⇒ Serial (Filtered (Syntax ℓ)) where
  series          = fmap convert <$>^ series
  seriesWithEnv = fmap convert <$>^ seriesWithEnv
  -- Generating intermediate states
stepFor :: Language ℓ ⇒ Peano → Filtered (Syntax ℓ) → Filtered (State ℓ)
stepFor n x = x >>= aux n ∘ initState
  where aux Zero     = return
        aux (Succ n) = either (const emptyFilter) (aux n) ∘ step
instance (Convertable ℓ) ⇒ Serial (Filtered (State ℓ)) where
  series          = stepFor <$>^ series          <*>^ series
  seriesWithEnv = stepFor <$>^ seriesWithEnv <*>^ seriesWithEnv
  -- Generating final values
joinMaybe :: Filtered (Maybe a) → Filtered a
joinMaybe (Filtered ps x)
   = Filtered (ps ++ [mkProperty (isJust x)]) (fromJust x)
instance (Convertable ℓ) ⇒ Serial (Filtered (Value ℓ)) where
  series          = (joinMaybe ∘ fmap (runFor 1000)) <$>^ series
  seriesWithEnv = (joinMaybe ∘ fmap (runFor 1000)) <$>^ seriesWithEnv
  -- Generating split states
instance (Convertable ℓ, Splitter ℓ) ⇒ Serial (Filtered (Split ℓ)) where
  series          = fmap (func_v split_v ()) <$>^ series
  seriesWithEnv = fmap (func_v split_v ()) <$>^ seriesWithEnv
```

Figure 8.14: Generic *Serial* instances for some *Supercompiler* types.

```
instance Language Core where
  data Syntax Core = CSyn {cSyn :: Core.Exp Int}
  data Value  Core = CVal  {cVal :: Core.Value Int}
  data State  Core = CSte  {cSte :: Core.State Int}
  initState = CSte ∘ Core.initialState ∘ cSyn
  step (CSte ⟨ Γ | (\•→ x)^t | [] ⟩)     = Left $ CVal $ Just (Γ, (\•→ x)^t)
  step (CSte ⟨ Γ | (Con c ps)^t | [] ⟩) = Left $ CVal $ Just (Γ, (Con c ps)^t)
  step (CSte s) = maybe (Left $ CVal Nothing) (Right ∘ CSte) (Core.step s)
instance Normaliser Core where
  isNormal = Core.isNormal ∘ cSte
instance Terminator Core where
  data History Core = CHst {cHst :: Core.History Int}
  emptyHistory = CHst []
  canContinue (CHst hist) (CSte s) = CHst <$> Core.canContinue hist s
```

Figure 8.15: Instances of *Language*, *Normaliser* and *Terminator* for *Core*.

```
instance Memoiser Core where
  data BindingIn  Core = MemoIn  Core.Binder [Core.Promise]
  data BindingOut Core = MemoOut (Map Core.HP
                                      (Core.FreeRefs, State Core))
  emptyIn = MemoIn (Core.HP (−1), Set.empty) []
  runBWriter (BWriter (MemoOut w) (CSyn x)) = CSyn x'
    where x' = Map.foldlWithKey lets x (Map.map lams w)
          lams (free, s) = Core.reconstructLambdas free
                             $ Core.rebuild $ join $ fmap cVal $ eval_e s
          lets y p x = (letrec • = Core.abstract p x in
                                   Core.abstract p y)^0
  memoise_v  cont = choose_v remember useReference_v cont
  residualise_v cont = testMe' "residualise_v" $ residualise (func_v cont)
```

Figure 8.16: Instance of *Memoiser* for *Core*.

on page 127 at the end of this chapter. Information about binding contexts is retained by a verified supercompiler so that *BoundSyntax* values can be evaluated for the testing of properties.

The $memoise_v$ verified function adds this detail, distinguishing the function from its equivalent in Section 5.3.4. The *remember* auxiliary function recalls previous binding information that matches the current state. If a matching binder is found, a reference is created using the *useReference_v* trusted combinator. Otherwise, control is passed to the continuation *cont*. The *runBWriter* function reconstructs an equivalent syntax tree with the wider binding context represented using let-expressions.

SPLITS Splits in this hybrid verified supercompiler reuse the *Core.Split* structure from Section 5.3.3. The *Splitter Core* instance, defined in Figure 8.17, creates a verified version of *Core.split* using the $testMe'$ combinator. The $applySubterms_v$ function needs to perform some coercion on the tag types as the hybrid supercompiler expects all core language objects to be tagged with integers.

In the Section 5.3.3 definitions of splitting, we never considered the semantic meaning of the *Core.Split* data type from Figure 5.7. We choose a very simple definition by which we evaluate all the subterms and reconstruct abstract syntax trees for their final values using *Core.rebuild*. These syntax trees are then inserted back into the split contexts.

FINAL STATES Finally, in Figure 8.18 we define a verified function for recovering a final state from a core language value. This definition completes the *Supercompiler Core* instance for a working supercompiler implementation.

8.6 TESTING THE SUPERCOMPILER

We can extract an implementation and supercompile a core program. Applying it to the map-fusion example used in Section 5.1, gives the same result as our reference implementation from Chapter 5.

```
 >>> func_v supercompile () (CSyn Core.mapmap)
CSyn {cSyn = letrec h0 = \f g xs ->
  case xs of { Nil       -> Nil;
               Cons y ys -> Cons (f (g y)) (h0 f g ys) }
in h0}
```

To complete the hybrid verification, we must now test the assumptions made by our supercompiler implementation. Before we can apply Lazy SmallCheck to our properties, a few further language-specific instances are required.

```
instance Splitter Core where
  data Split Core = CSpl {cSpl :: Core.HeapSplit Int}
  split_v = testMe' "split_v" $ const $
    λs → if isNormal s then CSpl ∘ Core.split ∘ cSte $ s
                       else CSpl ∘ Core.mkLeaf ∘ cSte $ s
  runSplit (CSpl (Core.Split ctx stm))
    = initState $ tagCoerce ctx $ fmap runRebuild stm
    where runRebuild = (Core.rebuild ∘ join ∘ fmap cVal ∘ eval_e ∘ CSte)
  applySubterms_v f = testMe' "applySubterms_v" $
    λr (CSpl (Core.Split ctx stm)) →
    (tagCoerce ctx ∘ fmap cSyn) <$> mapM (func_v f r ∘ CSte) stm
  -- Coerce tag types on split contexts
tagCoerce :: (Core.Tree (Exp ()) → Exp ()) →
             (Core.Tree (Exp Int) → Syntax Core)
tagCoerce ctx = CSyn ∘ Core.reTag ∘ ctx ∘ fmap Core.unTag
```

Figure 8.17: Instance of *Splitter* for *Core*.

```
instance Supercompiler Core where
  finalState_v = testMe' "finalState_v" $ const $
    CSte ∘ Core.initialState ∘ Core.rebuild ∘ cVal
```

Figure 8.18: Instance of *Supercompiler* for *Core*.

```
  -- Build a history from an intermediate state
mkHistory :: State Core → History Core
mkHistory = CHst ∘ aux
  where aux s = Core.stateTags (cSte s) : either (const []) aux (step s)
instance Serial (Filtered (History Core)) where
  series          = fmap mkHistory `fmap` series
  seriesWithEnv   = fmap mkHistory `fmap` seriesWithEnv
  -- Build a binding context from a list of states
mkBindingIn :: [State Core] → BindingIn Core
mkBindingIn []       = emptyIn
mkBindingIn (s : xs) = MemoIn this (promises' s')
  where
    MemoIn ((i, _)) promises = mkBindingIn xs
    (free', s') = Core.stripLambdas $ cSte s
    this = (pred i, free')
    promises' s = (this, s) : maybe promises promises' (Core.step s)
instance Serial (Filtered (BindingIn Core)) where
  series          = fmap mkBindingIn `fmap` series
  seriesWithEnv   = fmap mkBindingIn `fmap` seriesWithEnv
```

Figure 8.19: *Serial* instances for some *History Core* and *BindingIn Core*.

An instance *Convertable Core* defines a translation from *ProgGen.ProR* to *Syntax Core*. It is omitted here due to its length and the routine nature of the encoding.

The *Serial* instances for *Filtered* (*History Core*) and *Filtered* (*BindingIn Core*) are defined in Figure 8.19. Termination histories are generated by calculating the tags on chains of intermediate states. Binding contexts are created from sets of chains of intermediate states.

With this test-data generation apparatus in place, we can now query our implementation to test its assumptions using the *depthCheckAssumptions* combinator described in Section 8.2.4.

A peculiarity of Gill's (2009) *Data.Reify* combined with our heavy use of type-classes is that the verification program must be compiled to work.

Here, for example, is the output from executing our verification program, listed in Figure 8.20, which tests the supercompiler properties up to a depth 4.

```
main = depthCheckAssumptions 3
  (supercompile :: Syntax Core ↷ Syntax Core)
```

Figure 8.20: Execution of property-based testing on the supercompiler.

```
 $ time ./Chapter8
* Assumed correctness of Left
* Assumed correctness of Right
* Assumed correctness of initState_v
* Assumed correctness of runBWriter_v
* Assumed correctness of step_v
* Assumed correctness of useReference_v
* Semantic preservation over applySubterms_v
Passed after 482591 tests.
* Semantic preservation over finalState_v
Passed after 445984 tests.
* Semantic preservation over split_v
Passed after 445983 tests.
./Chapter8  488.71s user 4.72s system 99% cpu 8:15.20 total
```

The verification program lists six functions that have been assumed to preserve semantics using the *trustMe* combinator, where their correctness has been demonstrated through mechanised proof in Chapter 7. This is followed by three functions that have had their semantic preservation checked through property-based testing.

One may notice that absence of the $residualise_V$ verified function. This is due to the method by which we formulated the $applySubterms_V$ function in the *Splitter Core* instance, specifically with how the functional argument was handled. In the definition of $applySubterms_V$ in Figure 8.17 (on page 122), the verified continuation is unwrapped using the $func_V$ combinator, preventing the property from being collected by the verification algorithm. As it is here in the supercompilation algorithm (see Figure 8.10 on page 115), that $residualise_V$ is introduced, the property does not appear *independently* of that for $applySubterms_V$.

The hybrid verification framework does test the semantic preservation property for $residualise_V$. However, it is indistinguishable from the property testing $applySubterms_V$. The correct identification of this property should be `applySubterms_v residualise_v` but there is no facility for this in the current form of the framework. Either $applySubterms_V$ should be reimplemented such that it does not use $func_V$ to access the continuation or our hybrid verification model needs to be adjusted to properly identify partially applied functions.

To test the ability of the hybrid verification framework to find bugs, we inject a defect into our supercompiler implementation. The stack splitting algorithm from Figure 5.10 is adjusted so that the the application and update stack frame rules are swapped. Our hybrid verification process reports the following result:

```
...
* Semantic preservation over split_v
Failed after 104524 tests.

_
Succ (Succ (Succ Zero))
data D = A | B
f x = case x of {A -> B}
> f A
```

It correctly reports a fault in the $split_v$ function definition, which becomes apparent with the state obtained by running the above program for three steps.

Both a working supercompiler implementation and a set of correctness properties are mechanically extracted from a single program. Built on the foundation of a machine-checked proof, this is the hybrid verification of a supercompiler.

8.7 SUMMARY

Our supercompiler verified using a hybrid method is composed of several components. In Chapter 7, we showed that if each of these components is verified in isolation, then the whole compiler must be semantics preserving. Some of the components, such as normalisation and termination are shown to be correct through mechanised proof using properties-as-types. Others are shown to be correct through property-based testing.

The method by which we integrate these results is through the *verified function* generalisation, originally introduced in Section 7.4 but developed further in Section 8.2. We use this structure to mechanically collect the properties that need to be tested to show that the supercompiler is correct. These properties are then systematically tested using Lazy SmallCheck.

Our results show that verified functions provide a natural interface for building compilers that are correct-by-construction. The verified construction follows a functional programming style that ensures any problems that would be identified by property-based testing are located to a particular component and as more rigorous verification is performed on components, property-based testing can be replaced with mechanised proof. The next chapter, especially Section 9.6, discusses several extensions to these techniques.

```
instance Category (Verified r) where
  id  = IsVerified (const id) Map.empty
  (∘) = Comp
  -- Combinators about sums
infixr 2 |||
(||||) :: (r ⇝ a ↷ c) → (r ⇝ b ↷ c) → (r ⇝ Either a b ↷ c)
f ||| g = Merge f g
left_v :: (r ⇝ a ↷ Either a b)
left_v = trustMe "Left" $ const Left
right_v :: (r ⇝ b ↷ Either a b)
right_v = trustMe "Right" $ const Right
  -- Conditionals based on a property-irrelevant argument
choose_v :: (r → a → Either r0 r1) →
            (r0 ⇝ a ↷ b) → (r1 ⇝ a ↷ b) → (r ⇝ a ↷ b)
choose_v m f g = Modify m (Either f g)
  -- Boolean specialisation of conditional
cond_v :: (r → a → Bool) → (r ⇝ a ↷ b) → (r ⇝ a ↷ b) → (r ⇝ a ↷ b)
cond_v f = choose_v (λr x → if f r x then Left r else Right r)
  -- Maybe specialisation of conditional
maybe_v :: (r → a → Maybe r') →
           (r ⇝ a ↷ b) → (r' ⇝ a ↷ b) → (r ⇝ a ↷ b)
maybe_v f = choose_v (λr → maybe (Left r) Right ∘ f r)
  -- Ignore property-irrelevant argument
const_v :: (a ↷ b) → (r ⇝ a ↷ b)
const_v = Modify (const $ const ())
  -- Modify property-irrelevant argument
modify_v :: (r → a → r') → (r' ⇝ a ↷ b) → (r ⇝ a ↷ b)
modify_v = Modify
  -- Supply property-irrelevant argument
apply_v :: (r ⇝ a ↷ b) → r → (a ↷ b)
apply_v f r = Modify (const $ const r) f
```

Figure 8.21: Arrow-like combinators for verified functions.

```
  -- Recall previous binding information
remember :: (BindingIn Core, a) → State Core →
            Either (BoundSyntax Core) (BindingIn Core, a)
remember (MemoIn this prevs, r) (CSte s)
    | null matches = Right (MemoIn this ((this, s) : prevs), r)
    | otherwise = Left $ head matches
  where matches =
    [ BWriter (MemoOut (Map.singleton iOld (freeOld, CSte sOld))) $
      CSyn $ foldr aux (Var (Fre iOld))^0 freeOld
    | ((iOld, freeOld), sOld) ← prevs
    , freeMapping ← maybeToList (sOld `Core.equivState` s)
    , Map.keysSet freeMapping ≡ freeOld
    , let find p = Map.findWithDefault p p freeMapping
    , let aux p x = (x ⊔ (Fre (find p)))^0 ]

  -- Use reference to binder
useReference_v :: (BoundSyntax Core) ⇝ State Core ↷ BoundSyntax Core
useReference_v = trustMe "useReference_v" $ const

  -- Create a new binder
type Phase = (BindingIn Core, History Core) →
             State Core → BoundSyntax Core

  -- Create a new binder, run continuation and then create binding syntax
residualise :: Phase → Phase
residualise cont (MemoIn (i, _) prevs, r) s
    | i ∈ ctx    = BWriter (MemoOut $ Map.delete i ctx)
                           (CSyn $ (letrec • = · in ·
                             (Core.abstract i (cSyn x'))
                             (Core.Scope (Var (Bnd (Ix 0)))^0))^0)
    | otherwise = BWriter (MemoOut ctx) x'
  where
    i' = pred i
    (ps', s') = Core.stripLambdas $ cSte s
    BWriter (MemoOut ctx) x' = cont (MemoIn (i', ps') prevs, r) (CSte s')
```

Figure 8.22: Auxiliary functions for the *Memoiser* component.

Part IV

CONCLUSIONS AND FURTHER WORK

9 CONCLUSIONS AND FURTHER WORK

This chapter reviews the results of the programme of research reported in this thesis. We first revisit our initial motivations from Chapter 1 and the literature reviews of Chapters 2 and 6. We then discuss, in successive sections, what contributions have been made in previous chapters. Each section closes with a brief discussion of areas for future research. The chapter closes with some final overarching conclusions.

9.1 INTRODUCTION

Previous chapters of this thesis have discussed techniques and technologies to assist in the *verification of compilers*. In particular, the verification of a *super-compiler* was introduced in Chapter 5 as a case study. The motivation for this programme of research stemmed from the apparent absence of formal verification methods from the development of popular compiler projects. The neglect of verification is in spite of the regular discovery of defects that invalidate semantic preservation between compiler inputs and outputs.

There have been many past endeavours to introduce *mechanised theorem proof* to compiler verification. In Chapters 1 and 6 we cited McCarthy and Painter's 1967 publication as the origin of compiler verification through hand proof. In Chapter 6 we observed many attempts in the forty-five years that followed to simplify the task of proving compilers correct, mostly through proof abstraction, management and mechanised checking. This line of work culminated in Leroy's (2009) *CompCert*, a formally verified C compiler targeting PowerPC processors. We conjectured in Chapters 1 and 6 that mechanised theorem proof is still not widely accepted due to the amount and nature of effort required in its application. It is not merely the experiences of Leroy's (2009) *"estimated 2-person years of work"* but also the often monolithic and non-incremental nature of the proofs.

An alternative to theorem proving, introduced in Chapters 1 and 2 is to apply *property-based testing* to the problem of compiler verification. There are several instances (Katayama, 2007; Palka et al., 2011; Pike et al., 2012) where property-based testing has been used to discover defects in what Leroy would call *"realistic compilers"*. However, we do not see the wide adoption of these techniques either. This could be due to the difficulties in automatically generating test programs and formulating properties for compiler verification.

In Chapter 1, we proposed the hybridisation of mechanised theorem proof with property-based testing for the purposes of compiler verification. The aim was to combine the automation and re-usability of property-based testing with a formal, trusted foundation provided by theorem proof.

This concluding chapter revisits various contributions set out in earlier chapters of the thesis, where each section discusses lessons learned and avenues for further work. Section 9.2 presents conclusions from the work performed on improving Lazy SmallCheck's functionality. Section 9.3 discusses our findings from experiments in using Lazy SmallCheck to generate valid and canonical programs as test-data. Section 9.4 summarises our review of supercompilation and a literate implementation for a small core language. Section 9.5 discusses the abstract formal model of a supercompiler developed as a basis for our hybrid verification technique. Section 9.6 discusses the results of combining property-based testing with the formal model to produce a verified supercompiler. Section 9.7 revisits our aims from Chapter 1 and discusses the overarching conclusions from this programme of research.

9.2 ADVANCES IN LAZY SMALLCHECK

Lazy SmallCheck (Section 2.3 and Chapter 3) is a property-based testing library that exhaustively searches a bounded space of test-data values for counter-examples to given properties about programs. We have contributed something missing in the original implementation (Runciman et al., 2008), testing properties containing existential quantification and functional values using a new implementation of the pruning strategy provided by Lazy SmallCheck.

Section 3.5.5 discussed the deep link between partially-defined functional values and partially-defined instances of the trie representation used within Lazy SmallCheck. Our strategy naturally supplies partial functional values using the non-strict semantics of the host language. Other property-based testing libraries (Claessen, 2012) need to explicitly express partiality in their internal representation.

We also showed the benefits of displaying partial counterexamples, where the original implementation would only return total counterexamples to properties. All the test examples in Section 3.2 returned counterexamples that succinctly express partial terms, aiding the tester in debugging software.

The main conclusion of this work is there are benefits to including partial values in the test-data for property-based techniques beyond the pruning of the test-data space. Both in the concise display of counterexamples and the generation of functional values, partial values play a crucial role in helping developers produce high-assurance software.

FURTHER WORK One could envision adding partial values to QuickCheck's (Claessen and Hughes, 2000) *shrinking* mechanism, which attempts to find smaller variants of the counterexample found through random search through the test-data space. These partial counterexamples are simpler in comparison with those produced by some other property-based testing libraries which show information irrelevant to the property being testing. Adding partial values should not impact runtime performance directly but the mechanism for displaying these partial values is more complicated, as discussed in Section 3.3.1.

Parallelisation of the refutation algorithm is a current area of investigation. A prototype implementation shows near-linear speedups, in multicore shared-memory environments, for benchmarks in which no counterexample is found. This benefit is derived from the tree structure of the Lazy SmallCheck test-value search space. However, in some benchmarks where a counterexample is found the overheads of continued searches in other threads can cause slowdowns rather than speedups. The efficient exploitation of what should be an *'embarassingly parallel problem'* (Moler, 1986) would be very advantageous for managing the large spaces of test-data for a supercompiler.

9.3 PROGRAMS AS TEST-DATA

The method by which test-data is generated is critical to the use of property-based testing in compiler verification. In Chapter 2 we discussed the previous work on generating programs as test-data by directly building only well-formed, well-scoped and well-typed programs.

The approach we presented in Chapter 4 instead produces all possible abstract syntax trees and uses declarative conditions to filter away any trees that do not satisfy the requirements for valid test programs. This approach allows simpler and more composable definitions than if we had directly generated only valid programs. We further apply the approach to generate only *canonical* programs (Section 4.3), reducing the number of tests that need to be performed.

The Lazy SmallCheck property-based testing library handles these declarative conditions well, efficiently pruning large classes of invalid and non-canonical programs, as shown in Section 4.4. The ability to display partial counter-examples (introduced in Chapter 3) is unnecessary in this instance as only fully-defined programs satisfy the validity conditions. Therefore, no partial-programs can invalidate the property to be used as a counterexample. However, in Section 4.5 we show that even small total examples can be useful for exploring the behaviour of compilers.

Using declarative predicates to describe relevant test-data programs seems far more natural than attempting to generate the required programs directly. We show that desired characteristic can be represented as separate predicates

that compose easily. This is not possible in the other approaches discussed in the chapter.

FURTHER WORK Each canonical test program represents a class of programs performing equivalent computations but with different naming, ordering or abstraction boundaries, or with redundant parts. We would like to verify that every interesting test program has a canonical equivalent. The program generating framework itself could be used to test the existence of canonical representatives. Assuming that every program is represented by a canonical variant, we could write a function to transform any given program into a canonical representative. We could then test whether the function satisfies its specification.

If every valid core program has an equivalent representative, then in that sense every core-program computation is represented in generated tests. This technique has proved very successful in reducing the exhaustive space of test programs. But what if some desired property of a compiler, or other program-processor under test, fails only when a program is in some way non-canonical? If only canonical programs are tested, such potential failures will go undetected. A solution is to attach a post-processor to the canonical program generator. Given each canonical program, the post-processor picks an equivalent at random, not forgetting the possibility of picking the canonical program itself.

The core language used in Chapter 4 lacks features found in other core representations of functional languages. Other core languages, such as *GHC External Core* (Tolmac et al., 2009) and *F-lite* (Naylor and Runciman, 2010), include primitive values and operations, (recursive) local definitions and higher-order functions. The abstract syntax data type, generator and validity checker could be extended to include these features. However, the search-space of generated programs would be greatly enlarged. Some further principles of canonicity would be needed to prune this space.

A simple method for generating test-programs that involve higher-order functions could introduce a fixed set of higher-order composition operators. We could then combine the functions from similarly typed test-programs, generated by the current method, using these composition operators. This would set up the kinds of fusion cases that supercompilation optimises well.

Although we have explained principles of canonicity in terms of our core language, the ideas are quite generic. In almost every programming language, or other complex structural representation, there are choices of names or positions, orderings and divisions between units, that do not fundamentally alter the computations or structures being described. There is also the possibility of parts that are in some sense redundant. So similar techniques might be applied successfully to generate test examples in quite different formalisms.

9.4 A REVIEW OF SUPERCOMPILATION

Chapter 5 introduced *supercompilation* as a potential case study for verification. It is selected because it is *(a)* a source-to-source transformation, *(b)* with close links to formal language semantics, but *(c)* has non-trivial control-flow. These features make it an interesting candidate for an investigation into compiler verification techniques.

We compared different implementations from the literature and, in Section 5.3, showed that a supercompiler can ultimately be decomposed into four distinct components. *(1)* A *normaliser* that simplifies terms. *(2)* A *terminator* that prevents non-terminating simplification. *(3)* A *splitter* that produces smaller terms for further supercompilation. *(4)* A *memoiser* that reuses the results of supercompiling equivalent terms.

We implemented a supercompiler along these lines for a small core language described in Section 5.2, using the process as an opportunity to contrast the approaches taken in other implementations. Ours most closely resembles that of Bolingbroke and Peyton Jones (2010), as their implementation makes explicit use of the operational semantics of the source language. Later, in Section 7.5.1, this tight relationship allowed us to generically verify the normalisation component for any language that fits an abstract verified model.

FURTHER WORK The reference implementation is, by intention, a simple supercompiler designed to give us the best chance of demonstrating correctness. However, the literature contains a wide range of extensions to our basic model. These deal with more advanced languages, improved efficiency of the supercompilation process and better performing residual programs resulting from supercompilation. These are essential techniques for making supercompilation an acceptable optimisation for practical programming languages.

9.5 THE PROOF OF AN ABSTRACT SUPERCOMPILER

Chapter 7 used Agda, a dependently-typed programming language, to express a *verified abstract supercompiler* that leaves the source language and operational semantics undefined. Any areas of a supercompiler that critically depends on these definitions are also left abstract but with clear verifiable interfaces on what is required for the full compiler to be correct.

These interfaces take the form of *verified functions over evaluable types*, functions that carry the proof that they maintain semantic correctness between inputs and outputs. The advantage of these constructs is that they provide modularity for proofs along the same boundaries as implementation and concisely identify areas for verification at the type-level.

Through this approach, we showed that several components such as normalisation and termination are semantics preserving *by construction* for any instantiation of the model and that if the other components are also semantics preserving, it follows that the whole supercompiler is correct. These results form the foundation of the hybrid verification framework developed in Chapter 8.

While it would be useful to apply automated program extraction to our Agda models, we found that the MAlonzo Agda-to-Haskell compiler does not produce code that is satisfactory for our needs. The Haskell programs produced have far greater lines of code and poorer runtime performance than their hand-written equivalents. We concluded that it would be better to translate our Agda model into Haskell by hand.

FURTHER WORK As discussed throughout Chapter 7 and, in particular, in Section 7.5.2, we have deliberately ignored proofs that the supercompiler must terminate for all inputs. Agda normally enforces totality of functions to ensure soundness of the logic. It was going to be very difficult to fit supercompiler termination proofs into Agda's requirements. There is recent work (Vytiniotis et al., 2012) on encoding proofs of termination based on well-quasi-orderings for the types of termination checkers used by mechanised constructive logics. This would be a good start towards demonstrating total correctness of the supercompiler. It would still remain to show that the amount of supercompilation left to do reduces with each split.

Although we had good reasons for the hand-translation approach taken in Chapter 8, it is unsatisfying to be missing a mechanically checked link between the proven Agda and efficient Haskell implementation. Given the unsuitability of the MAlonzo compiler, it would be necessary to find alternative translations. There is an Agda-to-Epic compiler currently in development (Fredriksson and Gustafsson, 2011) but Epic does not have the full features of the mature GHC Haskell implementation. Another alternative would be to reimplement our proof in Coq and use its more advanced Coq-to-Haskell compiler to produce efficient code.

9.6 THE HYBRID VERIFICATION OF A SUPERCOMPILER

In Chapter 8, we used the previous discussed techniques to produce a working, verified supercompiler for a core functional language. We translated the *verified functions over evaluable types* abstraction from Chapter 7 into Haskell and extended it to use property-based testing in place of properties as types.

The abstract supercompiler was translated into Haskell using type-classes and type families to represent postulated data types and functions. We also in-

troduced abstractions for generating test-data for the supercompiler correctness properties, using the techniques discussed in Chapter 4.

These interfaces were instantiated with components from the reference supercompiler implemented in Chapter 5 and properties were automatically collected by the hybrid verification framework. Through the systematic testing of the unresolved properties, we showed that the supercompiler is indeed correct for a set of canonical first-order programs as test-data. This set of programs was sufficient to find a counterexample to our correctness properties when a fault is injected into the supercompilation definitions.

FURTHER WORK If the property-based testing portion of our verified framework had reported a defect in the $applySubterms_V$ verified function, it would have been difficult to locate the issue due to the presence of higher-order functions in the memoisation and splitting components. Our current instantiations blur the boundaries between the two components making an error indistinguishable between the two. It is unclear if a revised instantiation can be resolve this issue within the current framework formulation or if extensions are required to the hybrid verification model to handle these cases.

Verified functions seem to form an arrow-like (Hughes, 2000) structure. If we were to complete the bi-catesian closed category instance, we would be able to make full use of *arrow notation* (Paterson, 2001) to ease the development of compilers with verified function over evaluable types. In particular, the `apply` and `curry` functions in the `Control.Arrow` library would give possible solutions to our issues with higher-order functions and simplify some of the implementation detail in the memoisation component.

One area that has not been discussed as a correctness property is that a supercompiler should improve, or at least not diminish, the performance of its input. It is not uncommon to find compiler 'optimisations' actually increasing program run-times but it is certainly unwelcome. To integrate *'do no harm'* property into the hybrid verification model, our semantic definitions of the evaluable types need to preserving the execution-time semantics of their inputs. Our current implementations do not make this requirement.

9.7 FINAL CONCLUSIONS

The stated aim from Chapter 1 was to collect evidence supporting the assertion that:

> Combining property-based testing and mechanised proof verifies compilers with higher confidence than property-based testing alone, for less effort than mechanised proof alone.

In the process of investigating this claim, I have significantly improved previously available tools for property-based testing and exploited the unique features of Lazy SmallCheck to generate relevant test-data and display concise counterexamples. The use of declarative predicates to define desirable test programs appears to be novel and well suited to property-based testing with test-directed pruning. This has great potential to assist in the verification of compilers, as a flexible technique that can be extended and composed as compiler specifications change.

Verified functions over evaluable types provide clear boundaries by which both compiler implementations and proofs can be modularised. This allows a divide-and-conquer strategy to compiler verification and appropriate verification techniques to be applied to different components of a compiler. This was used in our case study, the verified supercompiler.

Although my initial aims for this thesis have not been fully realised, progress have certainly been made to both the property-based testing and the mechanised proof of a supercompiler's correctness, using each method's strengths to support the other. It is my hope that this is beginning of an engineering methodology for building correct compilers, where proof obligations can be isolated and the most appropriate verification technique may be applied. This is the essential benefit of a hybrid verification.

BIBLIOGRAPHY

Andreas Abel. Specification and verification of a formal system for structurally recursive functions. In Thierry Coquand, Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs*, volume 1956 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2000.

Lennart Augustsson. A compiler for lazy ML. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 218–227. ACM, 1984.

Ralph-Johan R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, 1988.

Stefan Berghofer and Martin Strecker. Extracting a formally verified, fully executable compiler from a proof assistant. In *COCV'03, Compiler Optimization Meets Compiler Verification*, volume 82:2 of *ENTCS*, pages 377 – 394, 2004.

Dines Bjorner, Neil D. Jones, and Andrey Petrovych Ershov. Partial evaluation and mixed computation. In *Proceedings of the IFIP TC2 Workshop*. Elsevier, 1988.

Maximilian Bolingbroke. *Call-by-need supercompilation*. Ph.D. thesis, Computer Laboratory, University of Cambridge, May 2013.

Maximilian Bolingbroke and Simon Peyton Jones. Supercompilation by evaluation. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 135–146. ACM, 2010.

Maximilian Bolingbroke, Simon Peyton Jones, and Dimitrios Vytiniotis. Termination combinators forever. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 23–34. ACM, 2011.

Abdulazeez S. Boujarwah and Kassem Saleh. Compiler test case generation methods: a survey and assessment. *Information & Software Technology*, 39: 617–625, 1997.

Geoffrey L. Burn, Simon Peyton Jones, and John D. Robson. The spineless G-machine. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, LFP '88, pages 244–258. ACM, 1988.

Arthur Charguéraud. The Locally Nameless representation. *Journal of Automated Reasoning*, 49:363–408, 2012.

Jan Christiansen and Sebastian Fischer. EasyCheck — Test data for free. In Jacques Garrigue and Manuel V. Hermenegildo, editors, *Functional and Logic Programming*, volume 4989 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2008.

Koen Claessen. Shrinking and showing functions: *(functional pearl)*. In *Proceedings of the fifth ACM SIGPLAN symposium on Haskell*, Haskell '12, pages 73–80. ACM, 2012.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279. ACM, 2000.

Koen Claessen and John Hughes. Testing monadic code with QuickCheck. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 65–77. ACM, 2002.

Luís Cruz-Filipe and Bas Spitters. Program extraction from large proof developments. In David Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 205–220. Springer, 2003.

Haskell B. Curry and Robert Feys. *Combinatory Logic*. North-Holland, 1958.

Nils Danielsson and Patrik Jansson. Chasing bottoms. In *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 85–109. Springer, 2004.

Nils Anders Danielsson. Operational semantics using the partiality monad. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 127–138. ACM, 2012.

Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381–392, 1972.

Paul F. Dietz. The GCL ANSI Common Lisp test suite. URL: http://en.scientificcommons.org/42309664, 2008.

Edsger W. Dijkstra. Notes on structured programming. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*, pages 1–72. Academic Press, 1972.

David J. Duke, Rita Borgo, Malcolm Wallace, and Colin Runciman. Huge data but small programs: Visualization design via multiple embedded DSLs. In Andy Gill and Terrance Swift, editors, *Practical Aspects of Declarative Languages*, volume 5418 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2009.

Jonas Duregård, Patrik Jansson, and Meng Wang. Feat: Functional enumeration of algebraic types. In *Proceedings of the 2012 ACM SIGPLAN workshop on Haskell*, Haskell '12, pages 61–72. ACM, 2012.

Levent Erkök and John Matthews. High assurance programming in Cryptol. In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, CSIIRW '09, pages 60:1–60:2. ACM, 2009.

Andrey Petrovych Ershov. Mixed computation: potential applications and problems for study. *Theoretical Computer Science*, 18(1):41 – 67, 1982.

Olle Fredriksson and Daniel Gustafsson. A totally Epic backend for Agda. Master's thesis, Department of Computer Science and Engineering, Chalmers University of Technology, May 2011.

Andy Gill. Type-safe observable sharing in Haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 117–128. ACM, 2009.

John B. Goodenough. The Ada compiler validation capability. In *Proceedings of the ACM-SIGPLAN symposium on Ada programming language*, SIGPLAN '80, pages 1–8. ACM, 1980.

Thomas Hallgren. Haskell tools from the Programatica project. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, Haskell '03, pages 103–106. ACM, 2003.

Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(04):327–351, 2000.

Christian Holler. Grammar-based interpreter fuzz testing. Master's thesis, Department of Computer Science, Saarland University, June 2011.

William A Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda−Calculus and Formalism*, pages 479–490. Academic Press, 1980.

John Hughes. The design of a pretty-printing library. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*. Springer, 1995.

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

John Hughes. QuickCheck testing for fun and profit. In Michael Hanus, editor, *Practical Aspects of Declarative Languages*, volume 4354 of *Lecture Notes in Computer Science*, pages 1–32. Springer, 2007.

Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, Revised edition, 2012.

Neil D Jones. Automatic program specialization: A re-examination from basic principles. *Partial evaluation and mixed computation*, pages 225–282, 1988.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, 1993.

Peter A. Jonsson and Johan Nordlander. Positive Supercompilation for a higher order call-by-value language: Extended Proofs. Technical report, Luleå University of Technology, 2008.

Susumu Katayama. Systematic search for lambda expressions. In *Trends in Functional Programming Volume 6*, TFP2005, pages 111–126. Intellect Books, 2007.

Ilya Klyuchnikov. Supercompiler HOSC: proof of correctness. Preprint 31, Keldysh Institute of Applied Mathematics, Moscow, 2010.

Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

Pieter Koopman and Rinus Plasmeijer. Synthesis of functions using generic programming. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Approaches and Applications of Inductive Programming*, volume 5812 of *Lecture Notes in Computer Science*, pages 25–49. Springer, 2010.

Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. GAST: Generic automated software testing. In Ricardo Peña and Thomas Arts, editors, *Implementation of Functional Languages*, volume 2670 of *Lecture Notes in Computer Science*, pages 84–100. Springer, 2003.

Joseph B. Kruskal. Well-quasi-ordering, the Tree Theorem, and Vazsonyi's conjecture. *Transactions of the AMS*, 95:210–225, 1960.

Dimitur Nikolaev Krustev. A simple supercompiler formally verified in Coq. In *The Second International Workshop on Metacomputation in Russia*. Russian Academy of Sciences, 2010.

Dimitur Nikolaev Krustev. Towards a framework for building formally verified supercompilers in coq. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming*, volume 7829 of *Lecture Notes in Computer Science*, pages 133–148. Springer, 2013.

Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

David Lester. *Combinator Graph Reduction: A Congruence and its Applications*. Ph.D. thesis, Department of Computer Science, University of Oxford, 1988.

Pierre Letouzey. Extraction in Coq: An overview. *Logic and Theory of Algorithms*, pages 359–369, 2008.

Michael Leuschel. Homeomorphic embedding for online termination of symbolic methods. In Torben. Mogensen, David. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 379–403. Springer, 2002.

F. Lockwood Morris. Advice on structuring compilers and proving them correct. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, POPL '73, pages 144–152. ACM, 1973.

Anil Madhavapeddy, Yaron Minsky, and Marius Eriksen. CUFP 2011 workshop report. *Journal of Functional Programming*, 22:1–8, January 2012.

Simon Marlow. An extensible dynamically-typed hierarchy of exceptions. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, Haskell '06, pages 96–106. ACM, 2006.

Conor McBride and James McKinna. Functional pearl: I am not a number — I am a free variable. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 1–9. ACM, 2004.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. *Mathematical Aspects of Computer Science*, 19:33–41, 1967.

Erik Meijer. More advice on proving a compiler correct: Improve a correct compiler. Technical report, Utrecht University, 1994.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991.

Robin Milner. Implementation and applications of Scott's logic for computable functions. In *Proceedings of ACM conference on Proving assertions about programs*, pages 1–6. ACM, 1972.

Robin Milner and Richard Weyhrauch. Proving compiler correctness in a mechanized logic. *Machine Intelligence*, 7:51–70, 1972.

Sava Mintchev. *Machine-supported reasoning about functional language programs and implementations*. PhD thesis, Manchester, 1995.

Neil Mitchell. Rethinking supercompilation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 309–320. ACM, 2010.

Neil Mitchell and Colin Runciman. A supercompiler for core Haskell. In *IFL 2007*, volume 5083 of *Lecture Notes in Computer Science*, pages 147–164. Springer, May 2008.

Neil Mitchell and Colin Runciman. Losing functions without gaining data: Another look at defunctionalisation. In *Proceedings of the 2009 ACM SIGPLAN Symposium on Haskell*, Haskell '09. ACM, 2009.

Cleve Moler. Matrix computation on distributed memory multiprocessors. *Hypercube Multiprocessors*, 86:181–195, 1986.

Matthew Naylor and Colin Runciman. The Reduceron reconfigured. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 75–86. ACM, 2010.

Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2009.

Bryan O'Sullivan. The Criterion package, v0.5.1.1. URL: `http://hackage.haskell.org/package/criterion`, 2011.

Michal H. Palka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the sixth IEEE/ACM Workshop on Automation of Software Test*, AST '11, pages 91–97, 2011.

Will Partain. The nofib benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992*, Workshops in Computing, pages 195–202. Springer, 1993.

Ross Paterson. A new notation for arrows. In *Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 229–240. ACM, 2001.

Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1), January 2003.

Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Experience report: a do-it-yourself high-assurance compiler. In *Proceedings of the 17th*

ACM SIGPLAN International Conference on Functional Programming, ICFP '12, pages 335–340. ACM, 2012.

Jason S. Reich, Matthew Naylor, and Colin Runciman. Supercompilation and the Reduceron. In *The Second International Workshop on Metacomputation in Russia*. Russian Academy of Sciences, 2010.

Jason S. Reich, Matthew Naylor, and Colin Runciman. Lazy generation of canonical test programs. In Andy Gill and Jurriaan Hage, editors, *Implementation and Application of Functional Languages*, volume 7257 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 2012.

Jason S. Reich, Matthew Naylor, and Colin Runciman. Advances in Lazy SmallCheck. In Ralf Hinze, editor, *Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 53–70. Springer, 2013.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 37–48. ACM, 2008.

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 51–62. ACM, 2008.

Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7:231–264, 1997.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 1–16. ACM, 2002.

Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(06):811–838, 1996.

James W. Thatcher, Eric G. Wagner, and James B. Wright. *More on advice on structuring compilers and proving them correct*, volume 615 of *Lecture Notes in Computer Science*, pages 165–188. Springer, 1980.

Andrew Tolmac, Tim Chevalier, and The GHC Team. An external representation for the GHC Core Language (for GHC 6.10). URL: `http://www.haskell.org/ghc/docs/6.10.4/html/ext-core/core.pdf`, 2009.

Valentin Fyodorovich Turchin. A supercompiler system based on the language Refal. *ACM SIGPLAN Notices*, 14(2):46–54, 1979.

Dimitrios Vytiniotis, Thierry Coquand, and David Wahlstedt. Stop when you are almost-full. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, volume 7406 of *Lecture Notes in Computer Science*, pages 250–265. Springer, 2012.

Philip Wadler. Applicative style programming, program transformation, and list operators. In *Proceedings of the 1981 conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 25–32. ACM, 1981.

Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231 – 248, 1990.

Brian A. Wichmann and Arthur H. J. Sale. A Pascal processor validation suite. Technical Report CSU 7/80, National Physical Laboratories, Teddington, England, March 1980.