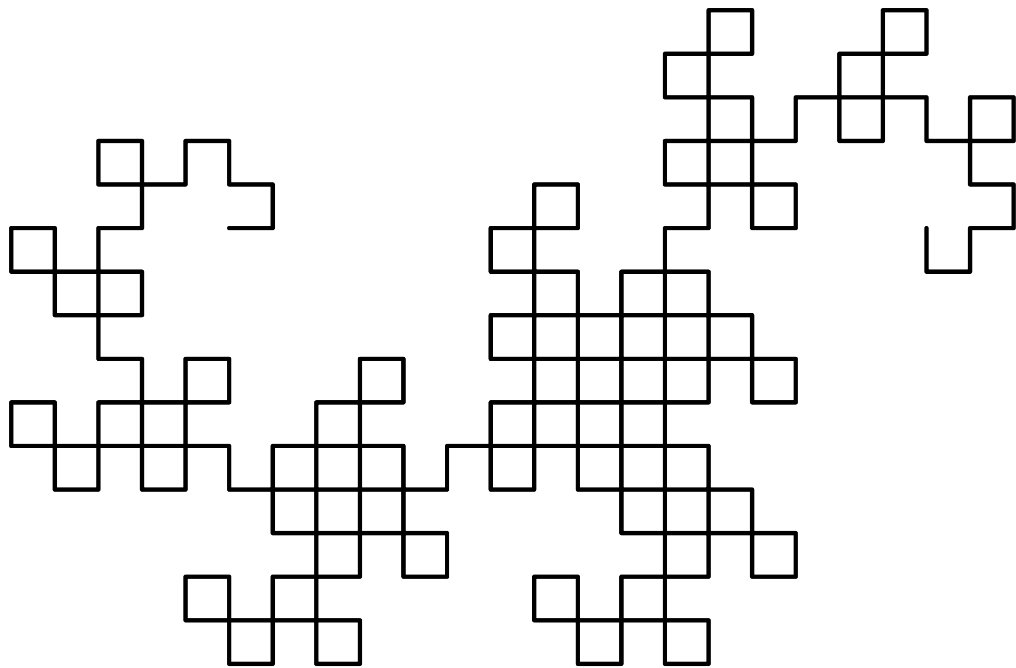
Random Term Generation for Compiler Testing in Spoofax

Master's Thesis



Martijn T. Dwars

Random Term Generation for Compiler Testing in Spoofax

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Martijn T. Dwars
born in Blaricum, the Netherlands

Programming Language Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

© 2018 Martijn T. Dwars.

Cover picture: The Harter-Heighway dragon (also known as “The Dragon Curve”) after eight iterations.

Random Term Generation for Compiler Testing in Spoofax

Author: Martijn T. Dwars
Student id: 4156730
Email: ikben@martijndwars.nl

Abstract

Testing is the most commonly used technique for raising confidence in the correctness of a piece of software, but constructing an effective test suite is expensive and prone to error. Property-based testing partly automates this process by testing whether a property holds for all randomly generated inputs, but its effectiveness depends upon the ability to automatically generate random test inputs. When using property-based testing to test a compiler backend, the problem becomes that of generating random programs that pass the parsing and analysis phase. We present SPG (SPoofax Generator), a language-parametric generator of random well-formed terms. We describe three experiments in which we evaluate the effectiveness of SPG at discovering different kinds of compiler bugs. Furthermore, we analyze why the generator fails to detect certain compiler bugs and provide several ideas for future work. The results show that random testing can be a cost-effective technique to find bugs in small programming languages (such as DSLs), but its application to practical programming languages requires further research.

Thesis Committee:

Chair: Prof. Dr. E. Visser, Faculty EEMCS, TU Delft
Committee Member: Dr. G. Gousios, Faculty EEMCS, TU Delft
Committee Member: Dr. R. Krebbers, Faculty EEMCS, TU Delft
Daily Supervisor: H. van Antwerpen, PhD Candidate, Faculty EEMCS, TU Delft

Preface

Ever since I learned to program I felt there was something magical about programming languages. However, it was not until I was introduced to the Programming Language group (which was then still the Software Language Design and Engineering group) that I became aware of Programming Languages (PL) as a research topic. This opened the door to a new and exciting world of formal semantics, type theory, program analysis and transformation, metaprogramming, and countless other topics that piqued my interest. Even though this thesis project is finished my interest for programming languages remains, and for this I am forever grateful.

I am thankful to all people that were involved with this thesis project. In particular, I would like to thank my supervisor Eelco Visser for suggesting the topic of random program generation. I would like to thank Hendrik van Antwerpen, Gabriël Konat, and Luís Eduardo Souza Amorim for their work on NaBL2, Spoofax Core, and SDF3, respectively, for without their effort, this work would not have been possible.

During my time at the PL group in Delft I met many inspiring people and without their help I would not have gotten where I am today. To my fellow students I would like to say that it has been a real pleasure to get to know you and share this time at the PL group with you.

Last, but certainly not least, I would like to thank my family for their love, support and constant encouragement I have gotten over the past seven years. Especially my parents deserve my wholehearted thanks for allowing me to realize my own potential. Without their help and endless amount of advice I would not be the person I am today.

Martijn T. Dwars
Delft, the Netherlands
December 12, 2017

Contents

Preface iii

Contents v

List of Figures ix

List of Tables x

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Research Questions . . . 3
- 1.3 Outline . . . 5

2 Preliminaries 7
- 2.1 Spoofax Testing Language . . . 7
- 2.2 Syntax Definition Formalism . . . 8
- 2.3 Program Transformation . . . 8
- 2.4 Name Binding Language . . . 10
- 2.5 Dynamic Semantics Language . . . 11
- 2.6 Type Soundness . . . 12

3 Sentence Generator 15
- 3.1 Motivation . . . 15
- 3.2 Kernel SDF and SDF3 . . . 16
- 3.3 Generation Algorithm . . . 17
- 3.4 Shrinking Algorithm . . . 19
- 3.5 More Shrinking Strategies . . . 20
- 3.6 Observing the Distribution . . . 20
- 3.7 Soundness and Completeness . . . 21

4 Sentence Generator Evaluation 23

4.1 Ambiguity Testing . . . 23
4.2 Observations . . . 25
4.3 Differential Testing . . . 26
4.4 Threats to Validity . . . 28

5 Term Generator **29**
5.1 Type System Specification with NaBL2 . . . 29
5.2 Term Generation Problem . . . 35
5.3 Generation Algorithm . . . 36
5.4 Term Generation Example . . . 41
5.5 Discussion . . . 44
5.6 Related Work . . . 47

6 Term Generator Evaluation **51**
6.1 Conformance Testing . . . 51
6.2 Type Soundness Testing . . . 53
6.3 Threats to Validity . . . 55

7 Analysis **59**
7.1 Generator Throughput . . . 59
7.2 Term Size . . . 61
7.3 Number of Resolutions . . . 61
7.4 Term Redundancy . . . 62

8 Discussion **63**
8.1 Algorithm Design Choices . . . 63

9 Related Work **67**
9.1 Type-Driven Generation . . . 68
9.2 Imperative Generation . . . 68
9.3 Needed Narrowing . . . 69
9.4 Enumeration . . . 70

10 Conclusions and Future Work **71**
10.1 Contributions . . . 71
10.2 Conclusions . . . 71
10.3 Future Work . . . 72
10.4 Source Code . . . 73

Bibliography **75**

A Grammar Differences **83**
A.1 Pascal . . . 83
A.2 Java 8 . . . 85

B Mutants **89**
B.1 L1-2-3 Mutants 89
B.2 Tiger Mutants 89

C Generated Programs **91**

List of Figures

2.1 Syntax test for L1 8
2.2 Syntax definition of L1 in SDF3. 9
2.3 ATerm language (simplified) 9
2.4 Algebraic signature of L1. 10
2.5 Subset of NaBL2 constraints. 11
2.6 Static semantics of L1 in NaBL2. 12
2.7 Dynamic semantics of L1 in DynSem. 13

5.1 Syntax of NaBL2 constraints 30
5.2 Interpretation of resolution and typing constraints 31
5.3 Resolution calculus 32
5.4 Example scope graph 33
5.5 Constraint solving algorithm 34
5.6 Syntax of L1 34
5.7 Static semantics of L1 35
5.8 Unification algorithm 38
5.9 Constructors and injections that are implicitly added to a signature. 40

6.1 Cumulative number of detected erroneous MiniJava compilers as a function of the number of executed tests. 52
6.2 Test case that exposes a bug in a compiler that was presumed to be correct. . . . 53
6.3 Mean time to kill mutants 1 to 6 with a 95% confidence interval. 54
6.4 Mean time to kill mutants 1 to 6 in the languages L1, L2, and L3 with a 95% confidence interval. 55
6.5 Mean time to kill mutant 1, 2, 4, 5, 8, and 12 with a 95% confidence interval. . 56

7.1 Average number of terms generated in one minute for L1, L2, and L3 with 95% confidence intervals. 60
7.2 Mean time to kill mutant M1-M6 in languages L1, L2, and L3 compensated for generator throughput. 60
7.3 Histogram of term sizes for a random sample of 1,000 generated terms in L3. . 61

7.4 Distribution of number of resolutions for 1,000 generated terms in L3. 62

List of Tables

4.1 The analyzed projects from MetaBorgCube. The first column contains the abbreviated git commit SHA-1 hash. The second and third column contain the time (in milliseconds) until an ambiguous sentence was found and the size (in characters) of the ambiguous sentence. The forth and fifth column contain the time it took to shrink the ambiguous sentence and the size of the shrunk sentence (in characters). No ambiguity could be found in the last four languages. . 24
4.2 Summary of the discovered differences between the SDF3 grammar and the ANTLRv4 grammar for Pascal and Java 8. 27

7.1 Group sizes and the frequency with which the group size occurs after grouping similar terms from a set of 1,000 generated terms in L1. 62

Chapter 1

Introduction

Software testing is the most commonly used technique of raising confidence in a program's correctness [8]. When testing a program, a programmer needs to construct a test input, run the program against the test input, and check the output for correctness. It is well recognized that software testing is expensive, sometimes accounting for approximately 50% of the cost of software development [44]. These costs are in part due to the development of the test suites and in part due to the maintenance of the test suites. The development of test suites is so costly because writing test cases is time-consuming, labour-intensive, and prone to human omission and error [36, 11]. The maintenance of test suites is so costly because test code, like production code, needs to be maintained, understood, and adjusted [21].

Much research has been devoted to automating the testing process [2]. Property-based testing alleviates the programmer from many of the aforementioned problems by automatically generating random test inputs and testing whether a property holds for each input. QuickCheck [32], the tool that popularized property-based testing, makes property-based testing possible by providing two domain-specific languages: one for defining properties to be tested and one for defining a test data generator. By default, a QuickCheck test data generator generates random values based on the type of the input, but it is also possible to define custom generators.

If the property to be tested includes a precondition, QuickCheck generates random values and filters values that fail to satisfy the precondition. This *generate-and-filter* approach is successful when the vast majority of the generated values satisfies the precondition. As the precondition becomes more complex, fewer test inputs will satisfy the precondition. Such preconditions are referred to as *sparse preconditions*. When too many test cases are discarded, property-based random testing becomes extremely slow. Moreover, the distribution of the generated test inputs might be skewed towards programs that fail to expose certain bugs. At this point, writing a custom generator is the only reasonable approach [25].

This situation naturally occurs when property-based testing is used to assess the correctness of a compiler. Due to the staged nature of a compiler, testing any stage of the compiler requires the test input to pass all preceding stages. For example, testing the backend of a compiler requires test inputs that pass the parsing stage. Similarly, verifying meta-theoretic properties such as type soundness requires test inputs that pass the type checker (i.e. programs that are well-typed). Unfortunately, a custom test data generator brings back much of

the problems that property-based testing tries to solve: developing a custom test data generator is time-consuming, labour-intensive, prone to human omission and error, and increases maintenance effort.

Formal specifications provide a precise and concise description of a system, making them an interesting tool to address the problem of generating valid test inputs. By having the specification drive the testing process, the test inputs are assured to be valid. Moreover, by deriving the test generator from the specification the creation of a separate test generator can be prevented, which reduces maintenance effort. In this thesis we explore and evaluate a technique to generate test-inputs based on the formal specification of a programming language. We first explore and evaluate a technique to generate valid sentences (strings in a formal language) based on the specification of the syntax of a programming language. We then extend this technique to generate well-typed terms based on the specification of a programming language's type system and name binding rules. This research is carried out in the context of the Spoofax language workbench, a platform for developing textual (domain-specific) programming languages. Both techniques are implemented as plugins for the Spoofax language workbench. The result is SPG (SPoofax Generator), a language-parametric generator of random sentences and well-typed terms in Spoofax.

1.1 Motivation

Testing is an important technique for validating and checking the correctness of software, but it is also difficult, expensive, laborious, error-prone and time consuming. Property-based testing alleviates a programmer from many of these problems. Indeed, property-based testing has seen much interest since the inception of QuickCheck [32]. Despite this increased interest, property-based testing has not yet reached mainstream adoption among compiler developers. This is unfortunate because many aspects of a compiler can potentially benefit from property-based testing. To motivate the use of property-based testing, and to get an impression of how property-based testing can help a compiler developer, we give an overview of current uses of property-based testing.

Testing Properties of Formal Models Formal verification is the act of proving or disproving the correctness of a system with respect to a certain formal specification or property using formal methods of mathematics. For example, many modern languages are proven to be *type sound*, which says that well-typed programs do not 'go wrong' at runtime. Type soundness is usually proven by separately proving the progress and preservation lemmas [48]. However, such formal proofs are such a large effort that they cannot be easily incorporated into the development workflow. In these cases, random testing can complement the use of formal methods by automatically exposing the system to a wide variety of inputs. For example, Fetscher et al. [19] test different languages for type soundness by generating and running well-typed programs provides a counterexample.

As another example, consider the problem of determining whether a context-free grammar is ambiguous. This problem is known to be undecidable, which means that which it is impossible to construct a single algorithm that always leads to a correct yes-or-no an-

swer. Moreover, searching for ambiguities in a grammar by hand is very hard due to the state explosion in context free grammars. In this case, testing for ambiguities provides a cost-effective alternative.

Besides type soundness and grammar ambiguity there are many other properties of models that can be tested. Klein et al. [29] demonstrate that with lightweight random testing tools they were able to discover bugs in nine ICFP papers.

Testing for Compiler Bugs Random testing has been used successfully to discover many bugs in C compilers. Lindig [36] tests the correct implementation of C's calling conventions amongst different C compilers on different platforms. Eide and Regehr [18] use random program generation to test the correct implementation of C's 'volatile' qualifier. Yang et al. [61] go even further by testing for arbitrary bugs in C compilers.

Testing Analysis Tools Refactoring is a technique for changing source code without changing its behavior. When developing refactoring algorithms one must make sure that the transformation does not change the semantics of the source code. Daniel et al. [13] show how random testing can be used to find bugs in the refactoring engines of Eclipse and NetBeans. Dewey et al. [16] test the correctness of a type checker by generating both well-typed and ill-typed programs. Midtgaard and Møller [42] use "quickchecking" to test a range of static analysis properties. They apply their technique to test the correctness of static analysis tools for the Lua programming language.

Testing a Compiler Backend The compilation process is traditionally organized as a sequence of stages where the input to each stage is the output of the previous stage. To assert the backend of a compiler, the test inputs need to get past the frontend of the compiler. For example, to test an optimising compiler, one would need generate well-typed programs, compile these with and without optimizations, and verify that the output is the same. Pałka et al. demonstrate how to test the optimizer of the Glasgow Haskell compiler by generating well-typed lambda terms [47]. Similarly, Mitgaardt et al. demonstrate how to use random testing to test two OCaml compiler backends against each other, successfully uncovering a number of bugs [41].

1.2 Research Questions

In this thesis we investigate the applicability and feasibility of random testing to raise confidence in a compiler's correctness. We first focus on the frontend of a compiler by investigating whether a random sentence generator can be used to detect ambiguities in a context-free grammar. There is a vast amount of literature on test input generation, but it is not obvious which techniques are suitable for program generation and how these techniques relate to the technologies used in Spoofax. To this end, we start by answering the following research question:

Research Question 1: *How to design and implement a language-parametric random sentence generator based on SDF3?*

Having designed and implemented a random generator of well-typed programs that is compatible with the Spoofax language workbench the next step is to evaluate the effectiveness of a random sentence generator at testing the correctness of a context-free grammar. To this end we answer the following research question:

Research Question 2: *How effective is a random sentence generator at discovering grammar ambiguities and grammar conformance?*

We then focus on the backend of a compiler by investigating whether a random term generator can be used to test type soundness as well as find bugs in the code-generation stage of a compiler. Testing the backend of a compiler requires the generation of well-typed terms which leads us to re-evaluate the previous research question for well-typed *terms* instead of valid *sentences*.

Research Question 3: *How to design and implement a language-parametric random term generator based on NaBL2?*

After having designed and implemented such a tool, we are interested in the effectiveness at discovering bugs in the backend of a compiler, which leads to the following research question:

Research Question 4: *How effective is the random program generator at discovering type soundness and code-generation bugs, respectively?*

The literature describes several techniques where random program generation is being used to test a compiler. However, these techniques are usually evaluated on toy programming languages such as the lambda calculus or small academic languages. These languages do not incorporate many features that are present in practical languages, such as type-dependent name resolution and subtyping. We would like to know how effective the generator is at discovering bugs in practical languages, which leads us to the fifth and final research question:

Research Question 5: *How does the generator scale as the language under test becomes more complex?*

1.3 Outline

This thesis consists of two parts. The first part (Chapter 3–4) presents our random sentence generation- and shrinking algorithm and evaluates its effectiveness at discovering grammar ambiguities and grammar differences. The second part (Chapter 5–6) presents our random term generation algorithm. Chapter **??** demonstrates the workings of the generator by generating a term step-by-step. Chapter 6 presents three case studies in which we evaluate the effectiveness of the term generator at discovering different kinds of compiler bugs in different languages. Chapter 9 reviews related work. Finally, Chapter 10 concludes this thesis and presents future work.

Chapter 2

Preliminaries

The Spoofax language workbench is a platform for the definition and development of textual (domain-specific) programming languages and IDEs [28]. Spoofax provides several meta-languages for specifying a language's syntax, static semantics, and dynamic semantics. Based on these specifications Spoofax generates a complete programming environment, including a parser, type checker, and interpreter.

This chapter briefly describes Spoofax's meta-languages that are important for SPG. We first describe SPT, Spoofax's language-parametric and declarative testing language. We then continue to describe NaBL2, Spoofax's language for specifying static semantics, and DynSem, Spoofax's language for specifying dynamic semantics. We illustrate each meta-language by showing an implementation of `L1`, a simple language with arithmetic expressions and first-class simply-typed functions [49].

2.1 Spoofax Testing Language

SPT (SPoofax Testing language) [26, 27] is a meta-language for the declarative definition of test cases for different aspects of a language, such as a language's syntax, static semantics, and dynamic semantics. SPT reduces the threshold for language testing by making it more convenient to write test cases. This is achieved by allowing test cases to be written using fragments of the language under test (the object language) instead of the language that is used for its implementation (the meta language). In addition, SPT provides several declarative constructs for specifying test expectations (assertions on tested fragments). For example, 'parse succeeds' asserts that the fragment can be parsed, 'resolve #*m* to #*n*' asserts that the identifier at marker *m* resolves to the identifier at marker *n*, and 'transform *x* to *y*' asserts that transforming the fragment using transformation *x* yields the term *y*. Because of the low threshold for writing tests SPT is particularly well-suited for testing a language as it is being developed. Figure 2.1 shows an example test case that tests that the addition of two integers is syntactically valid.

```
module syntax

language L1

test addition of two integers [[
  5 + 5
]] parse succeeds
```

Figure 2.1: Syntax test for L1

2.2 Syntax Definition Formalism

SDF3 (Syntax Definition Formalism) [57] is a meta-language for specifying a language's syntax. Like many grammar formalisms, SDF3 is based on the formalism of context-free grammars and its notation is similar to that of Backus–Naur Form (BNF). Similar to a context-free grammar, an SDF3 grammar consists of a set of production rules that describe all valid sentences (strings) in the formal language that it describes. However, SDF3 differs from context-free grammars in two important ways. First, the productions in an SDF3 grammar can be written as template productions in which whitespace is used to specify how the construct that is being described should be formatted [59]. Second, SDF3 is fully declarative: there is no target-language embedding for semantic actions. Instead, an SDF3 production can be labeled with a constructor that specifies the name of the term in the resulting Abstract Syntax Tree (AST). Based on an SDF3 definition Spoofax derives both a parser, a pretty-printer, and a first-order algebraic signature that describes the structure of well-formed ASTs.

Figure 2.2 shows the syntax definition of L1 in SDF3. The *context-free syntax* section specifies the grammar's high-level structure, whereas the *lexical syntax* section specifies the grammar's low-level structure [24]. To specify how to construct an AST, SDF3 allows the programmer to add a *constructor* to the production. For example, an application of the first production `Exp.IntV = INT` will result in an *IntV* term in the AST. An example of SDF3's declarative disambiguation constructs can be seen in the `left`, `right`, and `bracket` attributes. These attributes are disambiguation constructs that define the associativity of productions. Other disambiguation constructs, such as *context-free priorities* and *lexical restrictions*, have been omitted from this example for brevity.

2.3 Program Transformation

Stratego [58] is a program transformation language. The basic constructs in Stratego are *matching* a term against a pattern, denoted `?p`, and *instantiating* a pattern, denoted `!p`. Because the combination of matching a term against a pattern and instantiating a pattern is so common, Stratego provides the concept of a *rewrite rule*. A rewrite rule is of the form $r : p_1 \to p_2$ where r is the name of the rule and p_1 and p_2 are patterns. A rewrite rule

```
context-free syntax
  Exp.IntV  = INT
  Exp.Add   = <<Exp> + <Exp>> {left}
  Exp.Fun   = <fun(<ID>: <Type>) { <Exp> }>
  Exp.App   = <<Exp>(<Exp>)>
  Type.IntT = <Int>
  Type.FunT = <<Type> -\> <Type>> {right}
  Type      = <(<Type>)> {bracket}

lexical syntax
  ID     = [a-zA-Z] [a-zA-Z0-9]*
  INT    = "-"? [0-9]+
  LAYOUT = [\ \t\n\r]
```

Figure 2.2: Syntax definition of L1 in SDF3.

$$l := [t_1, \ldots, t_n] \mid [t \mid l]$$
$$t := c(t_1, \ldots, t_n) \mid l \mid \text{“...”}$$

Figure 2.3: ATerm language (simplified)

expresses that a term that matches the pattern p_1 can be replaced by an instantiation of the pattern p_2.

Whereas rewrite rules are ideal for describing local transformations, they are not suitable for describing global transformations. For describing global transformations Stratego provides higher-level (traversal) strategies. Such traversal strategies are parameterized with a transformation strategy and traverse the AST while applying the provided strategy at certain nodes. For example, `topdown(s)` applies the strategy `s` at every node in a top-down manner until `s` fails. The Stratego Standard Library (SSL) comes with a rich set of higher-order strategies for traversing terms in various ways.

Strategies operate on terms. A term is either an application of a constructor to a list of terms, denoted $c(t_1, \ldots, t_n)$, or a list of terms. A list of terms is written by enumerating its elements between square brackets, i.e. $[t_1, \ldots, t_n]$ or as the concatenation of a head and a tail, i.e. $[h|l]$. The syntax of terms is summarized in Figure 2.3.

The valid terms in the object language are described by an algebraic signature. Similar to how a context-free grammar describes all possible strings in a given formal language, an algebraic signature describes all possible ASTs. A signature declares a number of sorts and a number of constructors for these sorts, each having a name and zero or more terms of some sort. Formally, signatures are defined as follows.

```
signature
  constructors
    IntV : INT -> Exp
    Add  : Exp * Exp -> Exp
    Fun  : ID * Type * Exp -> Exp
    App  : Exp * Exp -> Exp
    IntT : Type
    FunT : Type * Type -> Type
```

Figure 2.4: Algebraic signature of L1.

Definition 1 (Algebraic Signature)**.** *An algebraic signature* Σ *is a pair* (S,C) *of a set of sorts* S *and a set of constructors* C*. A constructor* $c : s_1 \times s_2 \times \ldots \times s_n \rightarrow s_0$ *with name* c *takes* n *children of sorts* $s_1, s_n, ..., s_n$ *and is of sort* s_0 $(s_i \in S)$*.*

A constructor that takes zero children is called a *term injections*. Given an injection from s_j to s_i, denoted $: s_i \rightarrow s_j$, a term of sort s_i can be used whenever a term of sort s_j is expected. Spoofax automatically derives the algebraic signature from the SDF3 definition. The signature that is derived for the syntax definition of L1 is shown in Figure 2.4.

2.4 Name Binding Language

The static semantics of a programming language describes the meaning of the program at compile time. Most commonly, the static semantics describes the name binding structure and the type system of a programming language. Name resolution is the process of resolving tokens within program expressions to the intended program components. This process is complicated by scoping (e.g. lexical scoping), resolution to multiple declarations (such as with partial classes in C#), namespacing in many languages, types and variables live in different namespaces), overloading, and special restrictions. Type checking involves binding a program expression to a type and checking whether the assigned types follow the rules of the type system. For example, in many programming languages assignments are required to be type-compatible.

NaBL2 (Name Binding Language) is a meta-language for specifying a language's static semantics. NaBL2 uses a constraint-based approach to static analysis. First, a language-dependent traversal over the AST collects the constraints that must be satisfied for the program to be semantically valid. Among these constraints are constraints for building up a *scope graph*, a language-independent representation of the binding structure of a program [45]. Then, a language-independent solver is used to find a substitution that satisfies all constraints. From the constraints and corresponding substitution, all static analysis information (e.g. typing, name resolution) can be derived.

The language-dependent traversal over the AST is implemented as a set of *constraint generation rules*. A constraint generation rule is represented as $[\![p \wedge (s) : t]\!] := c$ where

Name	Notation
CGDecl	$x \longleftarrow s$
CGRef	$x \longrightarrow s$
CDirectEdge	$s \longrightarrow s'$
CResolve	$x \mapsto \delta$
CTypeOf	$x : t$
CGenRecurse	$[\![p \wedge (s) : t]\!]$

Figure 2.5: Subset of NaBL2 constraints.

p is an AST pattern, s is a scope, t is a type, and c is the set of constraints that should be derived. More formally, the constraint generation rules encode a (recursive) function parameterized by a scope and a type from AST terms to constraints. Figure 2.5 shows the name and notation of a subset of NaBL2 constraints. The declaration constraint $x \longleftarrow s$ and reference constraint $x \longrightarrow s$ add a declaration and a reference with name x to a scope s, respectively. A direct edge constraint $s \longrightarrow s'$ adds an l-labeled edge between scope s and s'. A resolve constraint $x \mapsto \delta$ represents the resolution of the reference x to the (unknown) declaration δ. A typing constraint $x : t$ assigns to the name x the type t. A recurse constraint $[\![p \wedge (s) : t]\!]$ represents the recursive invocation of the constraint generation rule on the term p parameterized by the scope s and the type t.

2.5 Dynamic Semantics Language

The *dynamic semantics* of a programming language defines the behavior of a program at runtime. For example, the semantics may define the strategy by which expressions are evaluated to values (such as eager evaluation or lazy evaluation) or the manner in which control structures conditionally execute statements. There are many ways of defining a language's dynamic semantics. A significant amount of academic research went into formal semantics of programming languages, which allow execution semantics to be specified in a formal manner. To such formalisms are operational semantics and denotational semantics.

DynSem (Dynamic Semantics Language) is a meta-language for the concise specification of the dynamic semantics of programming languages. DynSem supports the specification of the operational semantics of a language by means of statically typed conditional term reduction rules [55]. Spoofax automatically generates a Java-based AST interpreter based on a DynSem specification.

Figure 2.7 shows the dynamic semantics of `L1` in DynSem. DynSem uses *production rules* to describe the relation (reduction) from program terms to values. To carry around contextual information such as variable environments and heaps DynSem provides *semantic components*. The first rule in Figure 2.7 specifies that to reduce a `Program(e)`, the expression e should be reduced to a value v in an empty environment. The second and third rule are straightforward and specify how to reduce an `IntV` and `Add` term, respectively. The

```
init ^ (s) : ty := new s.

[[ IntV(_) ^ (_) : TInt() ]].

[[ Add(e1, e2) ^ (s) : TInt() ]] :=
  [[ e1 ^ (s) : TInt() ]], [[ e2 ^ (s) : TInt() ]].

[[ Fun(x, t, e) ^ (s) : TFun(t1, t2) ]] :=
  [[ t ^ () : t1 ]], [[ e ^ (s') : t2 ]],
  Var{x} <- s', Var{x} : t1, s' ---> s, new s'.

[[ App(e1, e2) ^ (s) : t2 ]] :=
  [[ e1 ^ (s) : TFun(t1, t2) ]], [[ e2 ^ (s) : t1 ]].

[[ IntT() ^ () : TInt() ]].

[[ FunT(t1, t2) ^ () : TFun(t1', t2') ]] :=
  [[ t1 ^ () : t1' ]], [[ t2 ^ () : t2' ]].
```

Figure 2.6: Static semantics of L1 in NaBL2.

fourth rule specifies that a function can be reduced to a closure that captures the current variable environment E. The fifth rule specifies that to evaluate the application of a closure to an arbitrary value $v1$ in the environment E, first the variable x needs to be bound to the value $v1$ and then e should be reduced to a value $v2$.

2.6 Type Soundness

To explain the relevance of DynSem to our work we first need to introduce the concept of *type soundness*. Many modern languages strive to be *type sound*, a property that is commonly summarized as "well-typed terms do not go wrong [48]." To make precise what it means to *go wrong*, we follow Wright and Felleisen [60] in viewing a static type system as a filter that selects well-typed programs from a larger universe of untyped programs. A partial function $eval : Programs \to Answers \cup \text{WRONG}$ defines the semantics of untyped programs. The result WRONG is returned for programs whose evaluation causes a type error. In this thesis, we adopt the simplest soundness property which states that well-typed programs do not yield WRONG:

Definition 2 (Weak Soundness)**.** *If* $e : \tau$ *then* $eval(e) \neq \text{WRONG}$.

One of our goals is to automatically test a language's *type soundness*. We achieve this by automatically generating well-typed programs (based on the NaBL2 specification) and evaluating these programs (based on the DynSem specification). If a well-typed program

```
Program(e) -init-> v
where
  E {} |- e --> v.

IntV(i) --> NumV(parseI(i)).

Add(NumV(i), NumV(j)) --> NumV(addI(i, j)).

E |- Fun(x, _, e) --> ClosV(x, e, E).

App(ClosV(x, e, E), v1) --> v2
where
  E  |- bindVar(x, v1) --> E';
  E' |- e --> v2.
```

Figure 2.7: Dynamic semantics of L1 in DynSem.

yields WRONG when evaluated, we have found a counterexample demonstrating that the type system is not sound with respect to the dynamic semantics.

Chapter 3

Sentence Generator

As a first step towards a generator of well-typed terms we developed a generator of random well-formed sentences, i.e. syntactically valid programs. Sentence generators are widely discussed in the literature [23, 50, 10, 7, 43]. However, despite this fact, there does not exist an off-the-shelf solution for the Spoofax language workbench. This is unfortunate, since a sentence generator can assist the language engineer in many different ways.

We begin this chapter by motivating the use of a random sentence generator. We continue by describing the algorithms that are used to generate and shrink random sentences. We evaluate the effectiveness of the generation- and shrinking algorithms at finding and shrinking ambiguities in a set of Spoofax languages. This evaluation led to the discovery of numerous bugs in the Spoofax language workbench which are discussed last.

3.1 Motivation

SDF is a syntax definition formalism based on context-free grammars [57]. A context-free grammar describes the strings in a language using production rules. Specifically, any sentence in the language is derivable by repeated application of the production rules. Such a derivation imposes a (hierarchical) structure on the sentence that is derived. This structure is referred to as a *parse tree* or *concrete syntax tree*. If a sentence in the language of the grammar has more than one parse tree, then the grammar is said to be *ambiguous*.

Traditional scanner and parser generators either do not accept any ambiguity or implicitly disambiguate ambiguous grammars. SDF, on the other hand, aims to provide a fully declarative definition of the syntax of a language [1]. To achieve this goal SDF does not have any implicit, or hidden, lexical or context-free disambiguation mechanisms. Instead, SDF provides specific features for the disambiguation of ambiguous grammars. Associativity and priorities are used to disambiguate ambiguous expression syntax. Follow restrictions and reject productions are provided to express lexical disambiguation rules. The language engineer is responsible for applying the appropriate disambiguation constructs, but doing so is not trivial [56].

Ambiguous grammars are undesirable because the parse tree of a sentence is often used to infer its semantics and an ambiguous sentence can have multiple meanings. For program-

ming languages this is almost always unintended and ambiguities are generally considered a grammar bug. Given the severity of this problem, one may hope for a procedure to test whether the grammar is ambiguous. Unfortunately, it is well-known that the problem of computing whether a grammar is ambiguous is undecidable. Moreover, due to the diversity of cause of ambiguity, it is hard for language engineers to spot an ambiguous grammar. However, there exists an easy alternative: testing. It is easy to generate a random sentence from the grammar. By parsing the generated sentence and checking whether more than one parse tree results, we can test if the sentence is ambiguous. Once such a sentence has been found, the grammar is known to be ambiguous.

Two other applications of a random sentence generator are testing the correctness of a parser and testing the correctness of a pretty-printer. A parser should always yield a parse tree or a parse error, and, in particular, it should never crash. By automatically exposing the parser to many different inputs, we hope to find an input on which the parser goes wrong. For a pretty printer, we expect that if t is the AST for some well-formed sentence s, then $t = parse(prettyprint(t))$. A program for which this condition does not hold demonstrates a bug in the pretty-printer. Again, by automatically exposing the pretty-printer to many different inputs, we hope to find an input for which the pretty-printer does not adhere to this contract.

3.2 Kernel SDF and SDF3

The generation algorithm operates on kernel SDF [57] extended with basic symbols (sorts, character classes, literals, regular operators). A kernel SDF grammar G consists of a list of productions P. Each production $p \in P$ is of the form $\mathcal{A} \rightarrow \alpha$ where $\mathcal{A}$ is a symbol and α is a list of symbols. We use $P_{\mathcal{A}}$ to denote the set of productions with left-hand side symbol $\mathcal{A}$. A symbol is either a sort (representing a non-terminal), a character class (representing a set of characters), a literal, or a regular operator applied to a symbol. A character class consists of zero or more character ranges. Finally, a character range is either a set of characters or a single character.

$G ::= [p_0, ..., p_n]$	(grammar)
$p ::= \mathcal{A} \rightarrow \alpha$	(production)
$\alpha ::= [s_0 ... s_n]$	(symbol list)
$s ::= S \mid cc \mid lit \mid s+ \mid s* \mid s?$	(symbols)
$cc ::= [cr_0 ... cr_n]$	(character class)
$cr ::= c\text{–}c' \mid c$	(character range)
$c ::= \mathrm{i}$	(character)

A production can have one or more annotations; of importance to our work are `reject`-annotations and `cons(`c`)` annotations. A `reject`-annotation specifies that any string that is

derivable by the production is rejected for the symbol on the left-hand side. The `cons(`c`)`-annotation specifies how the parse tree should be constructed.

SDF3 is a richer formalism than kernel SDF, but we limit ourselves to the differences that are relevant for our presentation. First, SDF3 makes a distinction between 'context-free syntax' and 'lexical syntax'. For productions that are defined as 'context-free syntax', every symbol in the right-hand side may be surrounded by layout. Second, SDF3 supports regular operators. Regular operators are applied to symbols and have the conventional notation and semantics: $\mathcal{A}$`?` denotes the optional presence of $\mathcal{A}$, $\mathcal{A}$+ denotes one or more repetitions of $\mathcal{A}$, and $\mathcal{A}$`*` denotes zero or more repetitions of $\mathcal{A}$. These constructs result in special AST constructs. Depending on whether $\mathcal{A}$ is present in the input, $\mathcal{A}$`?` yields either $\text{appl}(\text{none}, [])$ or $\text{appl}(\text{some}, [t])$ where t is the result of parsing $\mathcal{A}$. Parsing $\mathcal{A}$+ yields a list of one or more terms $[t_1, ..., t_n]$ and parsing $\mathcal{A}$+ yields a list of zero or more terms $[t_1, ..., t_n]$.

A normalization function transforms an SDF3 grammar into a kernel SDF grammar. Part of this normalization function is to merge productions in context-free syntax and lexical syntax. For a context-free production a special-purpose symbol `LAYOUT?` is inserted between symbols on the right hand side. Moreover, a symbol $\mathcal{A}$ in the lexical-syntax is renamed into $\mathcal{A}$`-LEX` and a symbol in the context-free syntax is renamed into $\mathcal{A}$`-CF`. A connection between the context-free and lexical syntax is made by adding an injection from $\mathcal{A}$`-LEX` to $\mathcal{A}$`-CF`.

3.3 Generation Algorithm

After our brief introduction of kernel SDF, SDF3, the normalization of SDF3 to kernel SDF, and the relation between grammars and abstract syntax trees we are ready to present a high-level description of the generation algorithm. The main idea behind the generation algorithm is to use the kernel SDF grammar to recursively generate a term that would be the result of parsing the given symbol. When multiple productions are available a production is chosen uniformly at random. The generated term is then pretty-printed. By generating terms instead of sentences we can rely on the pretty-printer to add whitespace when necessary.

There is one caveat: in kernel SDF, regular operators are not given special treatment. Instead, regular operators are treated as first-class citizens and can appear at all positions where a normal symbol can appear. To ensure the correct tree is constructed for regular operators, the generator treats regular operators other than normal symbols. For example, for the symbol $\mathcal{A}$`?` the generator produces either an application of the nullary constructor 'none' or an application of the unary constructor 'some' to a randomly generated term for the symbol $\mathcal{A}$.

We are now ready to present the algorithm at a detailed level. As became clear from our high-level description, the generator takes a symbol and returns a term. To ensure that the algorithm terminates we add a bound on the size of the term to generate, as is common practice with random testing. As a result, the generator may fail to generate a term for the given symbol bounded by the given size. For this reason, the function 'gen' takes a symbol and a size and returns a term wrapped in the 'maybe' monad. We use *do-notation* to simplify the presentation.

$$\text{gen}(\mathcal{A}\text{-}\texttt{LEX}, s) \rightarrow \text{str}(\text{genLex}(\mathcal{A}\text{-}\texttt{LEX}))$$

$$\text{gen}(\mathcal{A}?\text{-}\texttt{CF}, s) \rightarrow \begin{cases} \text{appl}(\text{none}, []) \\ t \leftarrow \text{gen}(A, \frac{s}{2}); \text{appl}(\text{some}, [t]) \end{cases}$$

$$\text{gen}(\mathcal{A}+\text{-}\texttt{CF}, s) \rightarrow \begin{cases} [] \\ h \leftarrow \text{gen}(A, \frac{s}{2}); t \leftarrow \text{gen}(\mathcal{A}*\text{-}\texttt{CF}, \frac{s}{2}); [h \mid t] \end{cases}$$

$$\text{gen}(\mathcal{A}*\text{-}\texttt{CF}, s) \rightarrow \begin{cases} [] \\ h \leftarrow \text{gen}(A, \frac{s}{2}); t \leftarrow \text{gen}(\mathcal{A}*\text{-}\texttt{CF}, \frac{s}{2}); [h \mid t] \end{cases}$$

$$\text{gen}(\mathcal{A}\text{-}\texttt{CF}, s) \rightarrow \begin{cases} \text{appl}(c, [t_0, ..., t_m]) & \text{if } p\{\text{cons}(c)\} \\ t_0 & \text{otherwise} \end{cases}$$

$$\text{where } p : \mathcal{A}\text{-}\texttt{CF} \rightarrow [s_0...s_n] \in P_{\mathcal{A}\text{-}\texttt{CF}},$$
$$[s'_0...s'_m] = \text{clean}([s_0...s_n]), [t_0...t_m] = [\text{gen}(s'_0)...\text{gen}(s'_m)]$$

For a context-free sort $\mathcal{A}$-`CF`, the generator chooses a random production $p : \mathcal{A}\text{-}\texttt{CF} \rightarrow [s_0...s_n]$. The symbols s_i are cleaned by retaining only $\mathcal{A}$-`CF` symbols thereby removing, for example, `LAYOUT?-CF`. It then tries to generate terms $t_0, ..., t_m$ for each right-hand side context-free symbol. If this succeeds the generator returns a term that depends on the presence or absence of a constructor annotation. If a constructor annotation is present, an application of the constructor to the generated terms for each right-hand side context-free symbol is returned. If the constructor annotation is absent we assume that $m = 1$ and the single generated term is returned. If this fails the generator backtracks and chooses a different random production for the current left-hand side. If no production yields a complete term, the generator returns `Nothing`.

The function 'genLex' generates a string for the given symbol. The auxiliary function '++' is used to concatenate strings.

$$\text{genLex}([cr_1 \dots cr_n]) \rightarrow \text{get}([cr_1 \dots cr_n], \text{random}(\text{size}([cr_1 \dots cr_n])))$$
$$\text{genLex}(\mathcal{A}) \rightarrow \text{gen}(s_i) \text{++} \dots \text{++} \text{gen}(s_n)$$
$$\text{where } \mathcal{A} \rightarrow [s_1, ..., s_n] \in P_{\mathcal{A}}$$

The function 'size' computes the size of a character class and is defined as follows:

$$\text{size}([c]) = 1$$
$$\text{size}([c - c']) = c' - c + 1$$
$$\text{size}([cr_1...cr_n]) = \sum \text{size}(cr_i)$$

The function 'get' gets the nth character from a character class and is defined as follows:

$$\text{get}([c], n) = c$$
$$\text{get}([c - c'], n) = c + n$$
$$\text{get}([cr_1...cr_n], n) = \begin{cases} \text{get}(cr_1, n) & \text{if size}(cr_1) < n \\ \text{get}([cr_2...cr_n], n) & \text{otherwise} \end{cases}$$

Finally, the auxiliary function 'random' takes an integer n and returns a random number from the range $[0,n)$.

3.4 Shrinking Algorithm

Once an ambiguous sentence has been found it should be reported to the language engineer. However, ambiguous sentences found this way suffer from several problems that make them ill-suited for communicating the ambiguity to the language engineer. First, the ambiguous sentences are often large and nonsensical, which makes it hard for the language engineer to pinpoint the source of the ambiguity. Second, the ambiguous sentences may contain many ambiguities that each have a different cause, which makes the generated sentence less suitable for communicating the ambiguity (such as in a bug report). Third, after finding an ambiguous sentence the language engineer may want to convert it into a unit test to prevent regressions. It is considered good practice for a unit test to be as small as possible while still exposing the bug that it tests for, which is not the case for large random sentences. To solve these problems we *shrink* the ambiguous sentence to a smaller ambiguous sentence before reporting it to the language engineer.

The idea underlying the shrinking algorithm is to replace a random sub-term *subTerm* of size *size* and sort *sort* by a smaller sub-term of the same sort. To obtain a smaller sub-term of the same sort the shrinker invokes the generator with the parameters *sort* and $size-1$. However, by replacing the sub-term we may have removed the ambiguity from the sentence. To determine if the new term is still ambiguous it is pretty-printed and parsed. An ambiguous parse means that we now have a strictly smaller ambiguous term which is then returned. Otherwise, the shrinker tries to replace the next random sub-term by a smaller term.

Algorithm 1 Shrinking algorithm

```
function SHRINK(term)
    subTerms ← subterms(term)
    for subTerm ← shuffle(subTerms) do
        sort ← sort(subTerm)
        size ← size(subTerm)
        newSubTerm ← GENERATE(sort, size − 1)
        if newSubTerm ≠ ⊥ then
            newTerm ← replace(term, subTerm, newSubTerm)
            parsed ← parse(prettyPrint(newTerm))
            if ambiguous(parsed) = ⊤ then
                return newTerm
            end if
        end if
    end for
    return term
end function
```

The pseudocode for performing a single shrink-step is shown in Algorithm 1. The algorithm refers to several auxiliary functions whose definition is straightforward. The function 'subterms' computes all subterms of the given term (including the term itself). The function 'shuffle' creates a random permutation of the given list using the Fisher–Yates shuffle algorithm. Each of the $n!$ permutation is equally likely to be returned. The function 'sort' retrieves the sort of the term which is stored in an annotation on the term. The function 'size' computes the size of the term, expressed in the number of constructor applications. The function 'replace' creates a new term that is equal to the first argument except that occurrences of the second argument are replaced by the third argument. The function 'parse' invokes the JSGLR parser for the language under test. The parser is configured to not perform error recovery or imploding, since these are not necessary for determining whether the parse was ambiguous. The function 'prettyPrint' invokes the Stratego strategy `pp-debug`, which is the entry point for the Spoofax-generated pretty-printer. The function 'ambiguous' traverses the parse tree (top-down and left-to-right) until it finds an 'amb'-term. It returns $\top$ if an 'amb'-term is found and $\bot$ otherwise. By applying the SHRINK function to its result until no smaller ambiguous term can be found we obtain a local minimum.

3.5 More Shrinking Strategies

The shrinking algorithm is capable of shrinking large sentences to smaller sentences that can be interpreted by a human. By generating a new term that is no larger than the old term the shrinker has the potential of quickly replacing a large subterm that does not contribute to the ambiguity by a significantly smaller term. However, in many cases the shrunk sentences can be shrunk further by making the shrinker more intelligent. To shrink a term t' further we added the following two strategies:

(a) A list of n terms $[t_1, ..., t_n]$ is shrunk by omitting one of the terms from the list. For a list of size n, this yields n lists of size $n-1$.

(b) If a constructor application $c(t_1, ..., t_n)$ of sort S has a descendant t' of sort S' and there is an injection $S' \to S$, the constructor application $c(t_1, ..., t_n)$ can be replaced by t'.

The motivation for strategy (a) is that it enables shrinking of lists that contain an 'amb'-node without having to generate a completely new list. For large lists, generating a completely new list is unlikely to retain the ambiguity and could lead to a lot of wasted work. Strategy (b) is motivated by the observation that for relatively small terms the generator may not be able to generate a smaller term, even when such a smaller term exists.

3.6 Observing the Distribution

A common problem with random testing is that the user has no insight into the kind of test inputs that the software under test is exposed to. If the test data is not well distributed, then conclusions drawn from the test results may be invalid and lead to a false sense of

confidence. QuickCheck [32] allows the programmer to incorporate code for making observations about the test data into the property being tested. After testing is complete a summary of the observations is shown to the user.

We take a similar approach by registering the length of the generated sentences. After testing is complete we show a frequency distribution of the length of the generated sentences. For example, after failing to find an ambiguity our tool may report the following table, which shows that 6636 sentences were generated of length 0-708 which is 67.5% of the total, etc.

```
No ambiguous sentence found after 10,000 terms (96,072 ms).

### Statistics ###
0-708: 6636 (67.5%)
708-1416: 2042 (20.8%)
1416-2124: 799 (8.1%)
2124-2832: 255 (2.6%)
2832-3540: 72 (0.7%)
3540-4248: 16 (0.2%)
4248-4956: 5 (0.1%)
```

3.7 Soundness and Completeness

When designing an algorithm it is important to ask is whether it is *sound* and whether it is *complete*. The sentence generation algorithm is not sound, i.e. the generator may generate sentences that do not belong to the language. For example, SDF3's `reject` annotation and SDF3's 'follow restrictions' may remove certain sentences from the language. The sentence generation algorithm is complete, as it generates any sentence in the language (ignoring layout) with non-zero probability.

Soundness and completeness are both desirable properties. However, for the goal of ambiguity testing and differential testing, these properties may not be practical. Unsoundness is not too much of an issue as long as the generator does not spend too much time generating illegal sentences. The Spoofax-generated parser can be used to parse the sentence and a sentences that cannot be parsed are simply discarded.

Chapter 4

Sentence Generator Evaluation

We evaluate the practical applicability of our sentence generator in two experiments. First, we use the sentence generator to find ambiguities in SDF3 definitions by generating sentences from the SDF3 grammar, parsing the generated sentence, and checking the parse result. Second, we use the sentence generator to expose differences between an SDF3 and ANTLRv4 grammar for the same language by generating sentences based on the SDF3 grammar and parsing them using the ANTLRv4 grammar.

4.1 Ambiguity Testing

We analysed sixteen projects from the MetaBorgCube[1] organization, a collection of language projects developed with Spoofax. Table 4.1 lists the projects that were used in our evaluation. Projects that were not compatible with Spoofax 2.4.0 were updated.

For our evaluation we generated no more than 1000 terms with a maximum size of 1000. We stopped as soon as an ambiguous sentence was found and reported the time it took to generate the ambiguous sentence (first column), the size of the ambiguous sentence (second column), the time it took to shrink the ambiguous sentence (third column), and the size of the shrunk sentence (fourth column). The size of ambiguous sentences is reported as number of characters.

4.1.1 Results

Out of the sixteen languages, thirteen languages contain an ambiguity. Each ambiguity was discovered without much effort: we ran the generator no more than one minute on each project. We noticed that for each language the author(s) invested some time in removing ambiguities by adding context-free priorities, which makes us believe that the ambiguities were not intended but the author simply forgot about these ambiguities. In general, given that a grammar should never be ambiguous, we can only explain this high number of ambiguities by a lack of proper tool support.

[1] `https://github.com/MetaBorgCube`

Project	Commit	Time	Size	Shrink Time	Shrunk Size
metaborg-units/appfunc	5e9574b	11	130	287	56
metaborg-units/mixml	5e9574b	90	149	103	54
metaborg-units/sml	5e9574b	188	1357	1538	78
metaborg-units/stsrm	5e9574b	253	194	40	50
metaborg-units/units	5e9574b	17	197	426	104
metaborg-calc	66df966	21	874	1055	35
metaborg-sl	93e8e47	22	723	403	87
metaborg-while	683e4d5	6	115	33	37
metaborg-js	fe1aabc	9	232	101	70
metaborg-typescript	35d7bc7	9	145	106	77
metaborg-tiger	1743d5c	28	996	307225	62
metaborg-pascal	c54654a	40	218	117	102
metaborg-llir	0b422ea	6	112	21	43
metaborg-smalltalk	a411cc2	113	161	38	67
metaborg-coq	e224d2c	17	274	68	67
metaborg-paplj	3516234	62	72	38	48
stratego/typed	f900b6e	492	519	154	202
sdf3-demo	db4cb2e	62	314	77	129
spoofax-jasmin	66411e1	102	679	347	273
grace	211c98e	414	1108	1478	117
java-front	71f436e	3118	146	110	110
java-front/java8	35107e7	26621	534	2912	130
metaborg-state-machine	6009c13	-	-	-	-
metaborg-jinja	f18dbe3	-	-	-	-
MiniJava	05a5fc7	-	-	-	-
QL2	072bbc3	-	-	-	-

Table 4.1: The analyzed projects from MetaBorgCube. The first column contains the abbreviated git commit SHA-1 hash. The second and third column contain the time (in milliseconds) until an ambiguous sentence was found and the size (in characters) of the ambiguous sentence. The forth and fifth column contain the time it took to shrink the ambiguous sentence and the size of the shrunk sentence (in characters). No ambiguity could be found in the last four languages.

To evaluate the effectiveness of our shrinking technique we measure for every ambiguous grammar the size of the ambiguous program, the size of the shrunk program, and the time it took to shrink the program. The size of the program is measured in number of characters and the time to shrink the program is measured in milliseconds.

4.2 Observations

While performing the experiments we made several observations that could guide future development of Spoofax.

- **Parser bug 1** In *metaborg-units.mixml* and *stratego/typed* we generated a sentence that triggers an `IndexOutOfBoundsException` when given to the parser. These bugs have been reported [2].

- **Parser bug 2** In *metaborg-sl* we generated a sentence that triggers a `FilterException` when given to the parser. This bug has been reported [3].

- **Pretty-printer bug 1** In *metaborg-units.sml* and *spoofax-jasmin* we generated a sentence s that could be parsed to a term t, but where the pretty-printed version $prettyprint(t)$ could not be parsed. A pretty-printer is correct when pretty-printing followed by parsing yields the original term [54]. Any sentence that violates this contract demonstrates a bug in the pretty-printer.

  In *metaborg-units.sml* the pretty printer did not add whitespace around a list separator. For example, the symbol `{ID "and"}+` describes a list of `ID`s separated by 'and' where each occurrence of 'and' may be surrounded by whitespace. For example, 'a and b and c' would be parsed successfully. Since the pretty-printer does not surround 'and' with whitespace, pretty printing the list `["a", "b", "c"]` yields the sentence `aandbandc` which could no longer be parsed.

  In *spoofax-jasmin* we noticed that when pretty-printing the '.deprecated' directive the string ends with a newline. However, the follow restrictions require that '.deprecated' is followed by a space or a tab. As such, the pretty-printed term could no longer be parsed.

- **Pretty-printer bug 2** The Spoofax-generated pretty-printer contains rewrite rules to allow pretty-printing of ambiguous sentences. Specifically, for every sort `S` the pretty-printer creates the strategy

  ```
  prettyprint-L-S:
    amb([h|hs]) -> <prettyprint-L-S> h
  ```

  to transform the amb-term into the first alternative it describes. This approach works fine when the amb-node takes the position of a constructor application. However,

[2] `http://yellowgrass.org/issue/Spoofax/226`
[3] `http://yellowgrass.org/issue/Spoofax/228`

when the amb-node is in the position of a list, the pretty-printer applies the strategy `pp-H-list` or `pp-V-list` to the amb-node. These strategies fail since they expect a list and not an amb-node and as a consequence the pretty-printer crashes. This bug has been reported [4].

- **Parse table generator bug** In *metaborg-pascal* we generated a sentence s that, when parsed, could not be pretty-printed, violating the condition that $\mathit{prettyprint}(\mathit{parse}(s))$ is equal to s modulo layout. Manual inspection showed that the `case-insensitive` attribute, which makes literals in template productions case-insensitive, was broken in the Java parse table generator. This bug has been reported [5].

- **Default project is ambiguous** Every new Spoofax project comes with a default implementation for parsing common constructs such as strings and layout (including single-line comments and multi-line comments). Roughly speaking, a string is defined as a sequence of characters enclosed within double quotes. The sequence of characters can be a backslash followed by a double quote or anything other than a newline or double quote. A single-line comment is defined as two slashes and reaches until the end of the line. Now the sentence `"\"//a"` can be parsed as:

  - the string `"\"` followed by a single-line comment `//a"`, or;
  - the string `"\"//a"`.

  Since many Spoofax languages use this definition without making any modifications, this ambiguity is present in many Spoofax languages. This bug has been reported [6].

- **Code completion introduces ambiguity** Recently, Spoofax added support for code completions using placeholders [14]. This is implemented by adding extra productions to the grammar of the object language. Specifically, for every sort S, a production $S = \$S$ is added ('$' is the default prefix symbol, but this can be configured). However, when $\$S$ could already be derived from sort S, this introduces an ambiguity. For example, in Java, an identifier is of sort `ID` and can start with a dollar. As such, `$ID` can be either a placeholder for the sort ID or an identifier named `$ID`. Consequently, the completions feature may unintentionally make an otherwise unambiguous grammar ambiguous and it becomes a responsibility of the language engineer to configure the completions feature to avoid ambiguities.

4.3 Differential Testing

Differential testing [39] attempts to detect bugs by providing the same input to different implementations of the same application and observing differences in their execution. We evaluate how effective our sentence generator is at discovering differences between an SDF3

[4] `http://yellowgrass.org/issue/Spoofax/231`
[5] `http://yellowgrass.org/issue/Spoofax/227`
[6] `http://yellowgrass.org/issue/Spoofax/230`

Language	Reference	Number of Differences	Number of SDF3 Bugs
Pascal	ISO 7185:1990	19	15
Java 8	JLS, Java SE 8	16	13

Table 4.2: Summary of the discovered differences between the SDF3 grammar and the ANTLRv4 grammar for Pascal and Java 8.

grammar and ANTLRv4 grammar for Pascal and Java 8 using differential testing. Specifically, we generate sentences based on the SDF3 grammar and parse the generated programs with the ANTLRv4 grammar. A sentence that cannot be parsed indicates that either the SDF3 grammar is too liberal or the ANTLRv4 grammar is too restrictive. That is, either the SDF3 grammar accepts more sentences than is should, or the ANTLRv4 grammar accepts fewer sentences than it should. Unfortunately, differential testing does not tell us which of these two is the case. To decide whether this difference is a bug in the SDF3 grammar or the ANTLRv4 grammar, we consult the ISO 7185:1990 standard and the Java Language Specification (JLS) 1.8 for Pascal and Java 8, respectively.

4.3.1 Results

The JLS 1.8 differentiates different kinds of expressions and not every expression is allowed within every context. For example, a `StatementExpression` may be used to form a statement. A `StatementExpression` includes assignments and method invocations, but excludes arithmetic expressions. Similarly, an annotation may contain a `ConditionalExpression`. A `ConditionalExpression` includes many of the traditional expressions (disjunction, conjunction, comparisons, etc), but excludes lambda expressions and assignments. The ANTLRv4 grammar correctly differentiates between these different kinds of expressions, but the SDF3 grammar fails to do so. Consequently, the SDF3 grammar allows certain kinds of expressions in a context where this should not be allowed. Because rewriting the SDF3 grammar would be a large engineering effort, we aborted the evaluation at this point.

Table 4.2 summarizes the found differences between the SDF3 grammar and the ANTLRv4 grammar for Pascal and Java 8. Using the sentence generator we were able to discover 19 differences between the SDF3 and ANTLRv4 grammar for Pascal and 16 differences between the SDF3 and ANTLRv4 grammar for Java 8. For a description of each difference, see Appendix A.

By relating the difference to the language specification we can mark a grammar difference as a bug in the SDF3 or ANTLRv4 grammar. Out of the 19 differences between the Pascal grammars, 15 are due to bugs in the SDF3 grammar and 4 are due to bugs in the ANTLRv4 grammar. Out of the 16 differences between the Java 8 grammars, 13 are due to bugs in the SDF3 grammar and 2 are due to bugs in the ANTLRv4 grammar. These bugs were found with very little effort, usually within several seconds of generating and parsing the generated programs.

4.4 Threats to Validity

For both experiments we should note that the generalizability of the outcomes, i.e. the external validity, may be jeopardized by our selection of grammars to evaluate. For our first experiment, we do not know how many ambiguities the grammars contained nor do we know how much effort has been put in removing ambiguities. If a grammar contains many ambiguities, the sentence generator is more likely to find an ambiguity. For our second experiment, we do not know how many differences between the SDF3 and ANTLRv4 grammars were missed. To mitigate these effects we collected and performed the evaluation on a corpus of 25 SDF3 grammars that were created by (graduate) students at Delft University of Technology.

Chapter 5

Term Generator

We begin this chapter with an explanation of NaBL2 along with an example of how NaBL2 is used to define the static semantics for a small programming language. We continue with a definition of the term generation problem and present an algorithm that generates random well-typed terms based on an NaBL2 specification. We then discuss the correctness of the algorithm and explain several choices that were made in its design. Finally, we compare the algorithm described in this chapter with two related algorithms that are described in the literature.

5.1 Type System Specification with NaBL2

NaBL2 takes a two-phase approach to static analysis: first, the constraints for a given AST are derived; next, a language-independent constraint solver computes a solution to the constraint system. If a solution exists, then the static analysis results (e.g. typing and name resolution) can be derived from the solved constraint system. If no solution exists, then the program is ill-formed and the constraint solver reports an error.

An NaBL2 specification describes the first phase of this two-phase process, that is, how to derive the constraints for a given AST. Specifically, the specification consists of a set of rules that encode a constraint derivation function. Each rule consists of a pattern, a scope, a type, and a set of constraints. Given a term to analyze, the term is matched against the pattern and the corresponding constraints are collected. The constraint derivation function may invoke itself recursively. Syntactically, a constraint derivation rule is represented as $[\![T \wedge (S) : \mathcal{T}]\!] := C$, where T is a pattern, S is a scope, $\mathcal{T}$ is a type, and C is a constraint that should be derived. We may refer to C as a set of constraints or a single constraint that is the conjunction of constraints; the two interpretations are interchangeable. The syntax of the constraints is shown in Figure 5.1 and their semantics is as follows:

- A *recurse constraint* $[\![T \wedge (S) : \mathcal{T}]\!]$ represents a recursive invocation of the constraint derivation function on the pattern T and is parameterized with the scope S and type $\mathcal{T}$. The recurse constraints can be interpreted as encoding a traversal over the AST, collecting constraints along the way.

- A *reference constraint* $R \longrightarrow S$ (resp. declaration-constraint $D \longleftarrow S$) describes a reference R (resp. declaration D) in scope S. An edge-constraint $S \longrightarrow S$ describes a directed edge from the left scope to the right scope. Reference-, declaration-, and edge-constraints together form a *scope graph*, a formal representation of the naming structure of a program.

- A *resolution constraint* $R \mapsto D$ specifies that a reference R needs to resolve to a declaration D. Typically, the declaration is unknown and specified as a variable δ. For example, $x_i^{\mathsf{R}} \mapsto \delta$ specifies that the reference x_i^{R} needs to resolve to an unknown declaration δ.

- An *equality constraint* $T_1 \equiv T_2$ specifies that two terms T_1 and T_2 should be syntactically equal. These constraints typically involve a variable. For example, $\tau_1 \equiv \mathsf{TInt}$ specifies that the variable τ_1 should have the value TInt.

- A *type declaration constraint* $D : T$ specifies that a declaration D has type T. When the type T is unknown a variable τ may be specified. For example, $x_i^{\mathsf{D}} : \mathsf{TInt}$ specifies that the declaration x_i^{D} has type TInt, whereas $x_i^{\mathsf{D}} : \tau$ specifies that x_i^{D} has some unknown type τ.

- The *association constraints* $D \longrightarrow\!\!\triangleright\, S$ and $D \rightsquigarrow S$ specify S as the associated scope of declaration D. The former constraint is typically used to connect the declaration (e.g. a module) of a collection of names to the scope declaring those names (e.g. the body of a module). The latter constraint is typically used to resolve a reference in a scope that is yet to be determined (i.e. S is a scope variable ς).

$$
\begin{aligned}
\mathrm{C} &:= \mathrm{C}^{\mathrm{G}} \mid \mathrm{C}^{\mathrm{Res}} \mid \mathrm{C}^{\mathrm{Ty}} \mid \mathrm{C}^{\mathrm{Rec}} \mid \mathrm{C} \wedge \mathrm{C} \\
\mathrm{C}^{\mathrm{G}} &:= \mathrm{R} \longrightarrow \mathrm{S} \mid \mathrm{D} \longleftarrow \mathrm{S} \mid \mathrm{S} \longrightarrow \mathrm{S} \mid \mathrm{D} \longrightarrow\!\!\triangleright\, \mathrm{S} \\
\mathrm{C}^{\mathrm{Res}} &:= \mathrm{R} \mapsto \mathrm{D} \mid \mathrm{D} \rightsquigarrow \mathrm{S} \\
\mathrm{C}^{\mathrm{Ty}} &:= \mathrm{T} \equiv \mathrm{T} \mid \mathrm{D} : \mathrm{T} \\
\mathrm{C}^{\mathrm{Rec}} &:= [\![\mathrm{T} \wedge (\mathrm{S}) : \mathrm{T}]\!] \\
\mathrm{R} &:= x_i^{\mathsf{R}} \\
\mathrm{D} &:= x_i^{\mathsf{D}} \mid \delta \\
\mathrm{S} &:= s \mid \varsigma \\
\mathrm{T} &:= c(T, \ldots, T) \mid \tau
\end{aligned}
$$

Figure 5.1: Syntax of NaBL2 constraints

5.1.1 Satisfiability of Constraints

A satisfiability relation $\vDash$ makes precise what it means for a constraint to be satisfied. The satisfiability relation is defined on ground resolution constraints C^{Res} and typing constraints C^{Ty} and is relative to a context $(\mathcal{G}, \psi)$, where $\mathcal{G}$ is a ground scope graph and ψ is a type environment. The overall definition of satisfaction for a program p is:

$$\phi(C^{G}_{p}), \psi \vDash \phi(C^{Res}_{p}) \wedge \phi(C^{Ty}_{p})$$

where $\phi(E)$ denotes the application of the substitution ϕ to all the variables appearing in E that are in the domain of ϕ. Figure 5.2 describes the satisfiability relation as a set of inference rules. The C-TYPEOF rule specifies that a typing constraint $d : T$ is satisfied if the type environment binds type T to declaration d. The C-RESOLVE rule specifies that a resolution constraint $x^{\mathsf{R}}_{i} \mapsto x^{\mathsf{D}}_{j}$ can be satisfied if the reference x^{R}_{i} resolves to the declaration x^{D}_{j} in the scope graph $\mathcal{G}$. The C-EQUALS rule specifies that an equality constraint $t_1 \equiv t_2$ is satisfied if the terms t_1 and t_2 are syntactically equal. The C-ASSOC rule specifies that an association constraint $d \rightsquigarrow S$ is satisfied if S is the associated scope of d.

$$\frac{\psi(d) = T}{\mathcal{G}, \psi \vDash d : T} \quad \text{(C-TYPEOF)}$$

$$\frac{\vdash_{\mathcal{G}} p : x^{\mathsf{R}}_{i} \mapsto x^{\mathsf{D}}_{j}}{\mathcal{G}, \psi \vDash x^{\mathsf{R}}_{i} \mapsto x^{\mathsf{D}}_{j}} \quad \text{(C-RESOLVE)}$$

$$\frac{t_1 = t_2}{\mathcal{G}, \psi \vDash t_1 \equiv t_2} \quad \text{(C-EQUALS)}$$

$$\frac{d \longrightarrow S}{\mathcal{G}, \psi \vDash d \rightsquigarrow S} \quad \text{(C-ASSOC)}$$

Figure 5.2: Interpretation of resolution and typing constraints

5.1.2 Name Resolution in Scope Graphs

The satisfiability of a resolution constraint $x^{\mathsf{R}}_{i} \mapsto x^{\mathsf{D}}_{j}$ depends on whether the reference x^{R}_{i} can resolve to a declaration x^{D}_{j} in the scope graph $\mathcal{G}$. A *scope graph* represents the naming structure of a program in a lexically scoped language. The nodes in a scope graph represent references, declarations, and scopes and the edges between two scopes in a scope graph represent the lexical nesting of scopes. The *resolution calculus* makes precise what it means to resolve a reference to a declaration in a scope graph. Informally, a reference resolves to a declaration if there is a path in the scope graph from the reference to the declaration, the

reference and the declaration have the same name, and the declaration is "closest" (i.e. not hidden by a closer declaration with the same name). If no such declaration exists, the reference is unresolved and if multiple such declarations exist, name resolution is ambiguous.

Formally, name resolution is specified using the inference rules in Figure 5.3. Rule (*E*) specifies that if there is an edge from scope S_1 to scope S_2, then there is an edge $\mathbf{E}(S_2)$ from S_1 to S_2. Rule (*I*) specifies that there is an empty path from a scope to itself. Rule (*T*) specifies that there is a path $s \cdot p$ from scope S_1 to scope S_3 if there is an edge s from S_1 to S_2 and a path p from S_2 to S_3. Rule (*R*) specifies that a declaration x_i^{D} is reachable from a scope S if there is a path p from scope S to scope S' and x_i^{D} is declared in scope S'. Rule (*V*) specifies that a declaration x_i^{D} is visible from a scope S if x_i^{D} is reachable from S with a path p and there is no reachable declaration x_j^{D} with a shorter path $p' < p$. Rule (*X*) specifies that a reference x_i^{R} resolves to a declaration x_j^{D} with a path p if x_i^{R} is a reference in scope S and x_j^{D} is visible from S.

The full theory of name resolution is more complex and involves, among other features, a well-formedness predicate on paths, labeled edges, a partial order on the labels, and named imports. We have omitted these details for brevity; a full presentation can be found in Antwerpen et al. [3].

$$\frac{S_1 \rightarrow S_2}{\vdash \mathbf{E}(S_2) : S_1 \rightarrow S_2} \quad \text{(E)}$$

$$\vdash [] : S \twoheadrightarrow S \quad \text{(I)}$$

$$\frac{\vdash s : S_1 \rightarrow S_2 \qquad \vdash p : S_2 \twoheadrightarrow S_3}{\vdash s \cdot p : S_1 \twoheadrightarrow S_3} \quad \text{(T)}$$

$$\frac{\vdash p : S \twoheadrightarrow S' \qquad x_i^{\mathsf{D}} \leftarrow S'}{\vdash p \cdot \mathbf{D}(x_i^{\mathsf{D}}) : S \rightarrowtail x_i^{\mathsf{D}}} \quad \text{(R)}$$

$$\frac{\vdash p : S \rightarrowtail x_i^{\mathsf{D}} \qquad \forall j, p' (\vdash p' : S \rightarrowtail x_j^{\mathsf{D}} \Rightarrow \neg(p' < p))}{\vdash p : S \mapsto x_i^{\mathsf{D}}} \quad \text{(V)}$$

$$\frac{\vdash x_i^{\mathsf{R}} \rightarrow S \qquad \vdash p : S \mapsto x_j^{\mathsf{D}}}{\vdash p : x_i^{\mathsf{R}} \mapsto x_j^{\mathsf{D}}} \quad \text{(X)}$$

Figure 5.3: Resolution calculus

It helps to think of scope graphs using their graphical representation. Figure 5.4 shows the graphical representation of an example scope graph with two scopes s_1 and s_2, one reference $\mathrm{a}_1^{\mathsf{R}}$ in scope s_1, and two declarations $\mathrm{a}_2^{\mathsf{D}}$ and $\mathrm{a}_3^{\mathsf{D}}$ in scopes s_1 and s_2, respectively. From

the perspective of a_1^R the declaration a_2^D hides the declaration a_3^D, because both declarations have the same name ‘a’ and a_3^D is more distant. Consequently, the reference a_1^R resolves to the declaration a_2^D along the path $\mathbf{D}(a_2^D)$.

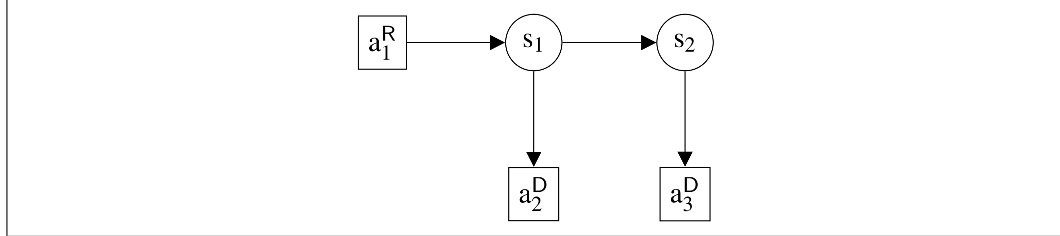


Figure 5.4: Example scope graph

5.1.3 Constraint Solving

After deriving the constraints for a given AST a constraint solver computes a solution to the constraint system. The constraint solving algorithm, originally presented in Antwerpen et al. [3], is shown in Figure 5.5. The algorithm is a non-deterministic rewrite system working over tuples $(C, \mathcal{G}, \psi)$ of a constraint, a scope graph, and a typing environment. The initial state of the solver is the collected constraint, the (incomplete) scope graph built from the scope graph constraints and an empty typing environment. The algorithm terminates when the constraint is empty or no more clauses can be solved. The rules in Figure 5.5 can be interpreted as follows:

- The rule S-ASSOC solves an association constraint $x^D \rightsquigarrow \varsigma$ by substituting the variable ς by a scope S if S is the associated scope of x^D.
- The rule S-EQUAL solves an equality constraint $T_1 \equiv T_2$ by unifying T_1 and T_2 and applying the resulting substitution to the tuple $(C, \mathcal{G}, \psi)$.
- The rules S-TYPEOF1 and S-TYPEOF2 solve a typing constraint $x^D : T$ by storing the fact that x^D has type T if no such fact exists and by adding a constraint that T should equal the stored type $\psi(x^D)$ otherwise.

5.1.4 Example NaBL2 Specification

To make clear how an NaBL2 specification is used to specify a programming language’s static semantics we next define the NaBL2 specification for L1, a simple language with arithmetic expressions and first-class simply-typed functions [49]. The syntax of L1 is shown in Figure 5.6. As expressions we include integer literals, variable references, additions, function abstractions, and function applications. As types we include the base type **Int** as well as the function type $t_1 \rightarrow t_2$.

$$
\begin{array}{lll}
(x^{\mathsf{R}} \mapsto \delta \wedge C, \mathcal{G}, \psi) \rightarrow & [\delta \mapsto x^{\mathsf{D}}](C, \mathcal{G}, \psi) \text{ where } x^{\mathsf{D}} \in R_{\mathcal{G}}(x^{\mathsf{R}}) & \text{(S-RESOLVE)} \\
(x^{\mathsf{D}} \rightsquigarrow \varsigma \wedge C, \mathcal{G}, \psi) \rightarrow & [\varsigma \mapsto S](C, \mathcal{G}, \psi) \text{ where } x^{\mathsf{D}} \twoheadrightarrow S & \text{(S-ASSOC)} \\
(T_1 \equiv T_2 \wedge C, \mathcal{G}, \psi) \rightarrow & \sigma(C, \mathcal{G}, \psi) \text{ where } \mathcal{U}(T_1, T_2) = \sigma & \text{(S-EQUAL)} \\
(x^{\mathsf{D}} : T \wedge C, \mathcal{G}, \psi) \rightarrow & (C, \mathcal{G}, \{x^{\mathsf{D}} \mapsto T\} \cup \psi) \text{ if } x^{\mathsf{D}} \notin \mathrm{dom}(\psi) & \text{(S-TYPEOF1)} \\
(x^{\mathsf{D}} : T \wedge C, \mathcal{G}, \psi) \rightarrow & (\psi(x^{\mathsf{D}}) \equiv T \wedge C, \mathcal{G}, \psi) \text{ if } x^{\mathsf{D}} \in \mathrm{dom}(\psi) & \text{(S-TYPEOF2)}
\end{array}
$$

Figure 5.5: Constraint solving algorithm

$$
\begin{array}{l}
e ::= n \mid x \mid e_1 + e_2 \mid \textbf{fun}(\mathrm{x} : t)\{e\} \mid e_1(e_2) \\
t ::= \textbf{Int} \mid t_1 \rightarrow t_2
\end{array}
$$

Figure 5.6: Syntax of L1

The NaBL2 specification for L1 is shown in Figure 5.7. The constraint derivation rules define the constraints that need to be derived for each syntactic construct. Moreover, the specification shows that the constraint derivation rules are invoked recursively and that the constraint derivation rules are parameterized with a scope parameter. The rules for L1 can be interpreted as follows:

- Rule 5.1 specifies that an integer literal has type Int.
- Rule 5.2 specifies that a variable reference in scope s needs to be resolved to a declaration δ of type t. The type of the variable reference is equal to the type of the declaration.
- Rule 5.3 specifies that an addition has type Int and that its operands also have type Int. The constraints for the operands are derived by recursively invoking the constraint derivation function.
- Rule 5.5 specifies that a function has type $\mathrm{Fun}(t_1, t_2)$, creates a new scope s' that inherits from scope s, and adds a declaration x_i^{D} to s' of type t_1. Note that the constraint derivation function is invoked recursively on terms t and e where the former invocation is parameterized with scope s whereas the latter invocation is parameterized with scope s'.
- Rule 5.6 specifies that the application of an expression e_1 to an expression e_2 has type t_2 where e_1 has type $\mathrm{Fun}(t_1, t_2)$ and e_2 has type t_1. The constraints for the subterms are derived by recursively invoking the constraint derivation function.
- Rule 5.7 specifies that a term **Int** has type Int.

- Rule 5.8 specifies that a term $t_1 \to t_2$ has type $\mathrm{Fun}(t'_1, t'_2)$ where the subterms t_1 and t_2 have types t'_1 and t'_2, respectively. The constraints for the subterms are derived by recursively invoking the constraint derivation function.

$$
\begin{aligned}
[\![n \wedge (s) : \mathrm{Int}]\!] &:= \mathsf{True}. && (5.1)\\
[\![x_i \wedge (s) : t]\!] &:= x_i^{\mathsf{R}} \longrightarrow s \;, x_i^{\mathsf{R}} \mapsto \delta, \delta : t. && (5.2)\\
[\![e_1 + e_2 \wedge (s) : \mathrm{Int}]\!] &:= [\![e_1 \wedge (s) : \mathrm{Int}]\!], [\![e_2 \wedge (s) : \mathrm{Int}]\!]. && (5.3)\\
[\![\mathbf{fun}(x_i : t)\{e\} \wedge (s) : \mathrm{Fun}(t_1, t_2)]\!] &:= \mathbf{new}\; s',\; s' \longrightarrow s \;,\; x_i^{\mathsf{D}} \longleftarrow s' \;, x_i^{\mathsf{D}} : t_1, && (5.4)\\
&\quad [\![t \wedge (s) : t_1]\!], [\![e \wedge (s') : t_2]\!]. && (5.5)\\
[\![e_1(e_2) \wedge (s) : t_2]\!] &:= [\![e_1 \wedge (s) : \mathrm{Fun}(t_1, t_2)]\!], [\![e_2 \wedge (s) : t_1]\!]. && (5.6)\\
[\![\mathbf{Int} \wedge (s) : \mathrm{Int}]\!] &:= \mathsf{True}. && (5.7)\\
[\![t_1 \to t_2 \wedge (s) : \mathrm{Fun}(t'_1, t'_2)]\!] &:= [\![t_1 \wedge (s) : t'_1]\!], [\![t_2 \wedge (s) : t'_2]\!]. && (5.8)
\end{aligned}
$$

Figure 5.7: Static semantics of L1

5.2 Term Generation Problem

Now that we have described the NaBL2 language and specified what it means for a constraint to be satisfied we are ready to define the *term generation problem*:

Definition 3. *The* term generation problem *is the problem of finding a program p for which the corresponding set of constraints* $[\![p]\!]$ *is satisfiable.*

Before we look at how we could solve the term generation problem we shift our focus towards a similar problem. The *type inhabitation* problem is the problem of determining whether a term with a given type exists (i.e. whether the type is 'inhabited'). Formally, the type inhabitation problem for a given calculus is the following problem: given a type τ and a typing environment Γ, does there exist a term M such that $\Gamma \vdash M : \tau$. Type inhabitation for the simply-typed lambda calculus is PSPACE-complete and hence decidable [53]. For more complex calculi, such as the polymorphic second-order and higher-order lambda calculi, the type inhabitation problem is undecidable. It is unclear whether the type inhabitation problem for arbitrary languages, such as a language defined by an NaBL2 specification, is decidable. If the type inhabitation problem for arbitrary languages is decidable, then this gives us a straightforward generation algorithm. It is our conjecture that this problem is undecidable. Under this assumption we cannot decide for an arbitrary language whether a term with a given type exists in a given context.

Instead, our goal is to create an efficient search algorithm that finds random solutions fast enough to make random testing feasible. One extreme is to repeatedly generate a syn-

tactically valid AST, derive the corresponding constraints, try to solve the constraints, and discard terms for which no solution exists (i.e. ill-formed terms). For practical programming languages, such a ‘generate-and-filter’ approach is likely to spend a lot of time generating and discarding ill-typed terms [31, 30]. This problem can be alleviated by solving constraints early on. However, generating a random program and solving the corresponding constraints in an incomplete program creates several challenges. For example, how should we generate a random program? When should we solve a constraint? In what order should we solve the constraints? What does it mean to solve a constraint in an incomplete constraint system? The next section presents a generation algorithm that solves these problems.

5.3 Generation Algorithm

To analyze a program, first all constraints are collected and then the constraints are solved to obtain the analysis results. The core idea underlying our generation algorithm is to alternate between collecting constraints and solving constraints. By solving constraints early on the generator can refine the program as it is being generated. For example, it may resolve references to declarations and assign types during generation, which guides the generator towards well-formed terms.

The generator starts with a program that consists of a variable pattern p, concrete scope s, variable type t, and a single recurse constraint $[\![p \wedge (s) : t]\!]$. If there are no constraints that need to be solved the program is returned. Otherwise, the generator creates a random permutation of the constraints and tries to solve the first constraint. If there are multiple solutions then the generator picks the first solution and continues to generate with this solution. If no constraint can be solved the generator backtracks. When backtracking, the generator first considers other solutions to a constraint. If no such solution leads to a complete program then the generator backtracks further and solves the next constraint.

Two mechanisms ensure that the generator terminates. First, the size of the generated term (counted as the number of constructor applications) may not exceed a predefined constant. As soon as the generated term becomes too large the generator backtracks, preventing the generator from generating an infinitely large term. Second, every time a constraint is solved a counter is incremented. When this counter reaches a predefined constant the generator aborts (i.e. returns without backtracking). This prevents the generator from spending too much time backtracking.

Algorithm 2 shows the pseudocode for the generation algorithm. The function GENERATE takes a program and returns a program in which all constraints are solved or $\perp$ if no such program could be found. It does so by solving a constraint, yielding zero or more programs in which the constraint is solved. However, the resulting programs may be inconsistent. The function *consistent* performs some lightweight checks to avoid that the generator continues generating with an inconsistent program (described in Section 5.3.3). Moreover, since resolution constraints are solved in an incomplete scope graph, the resulting program may invalidate the decision to resolve a reference to a declaration. The function *resolves* checks whether every reference can still resolve to the intended declaration (described in Section 5.3.4). Finally, the function GENERATE is invoked recursively with the resulting

program. The variables *maxSteps* and *maxSize* specify the maximum number of steps the generator can take before it should abort and the maximum size of the term to generate, respectively. The auxiliary function *shuffle* returns a random permutation of the given list, where every permutation is equally likely to occur.

Algorithm 2 Generation algorithm

```
function GENERATE(program)
    if steps > maxSteps then
        abort
    end if
    if size(program.pattern) > maxSize then
        return ⊥
    end if
    if program.constraints = ∅ then
        return program
    end if
    steps ← steps + 1
    for constraint ← shuffle(program.constraints) do
        solutions ← solver.solve(program, constraint)
        for solution ← solutions do
            if consistent(solution) ∧ resolves(solution) then
                complete ← generate(solution)
                if complete ≠ ⊥ then
                    return complete
                end if
            end if
        end for
    end for
    return ⊥
end function
```

The pseudocode shows that, to solve a constraint, the generator invokes the constraint solver. This constraint solver behaves the same way as the original constraint solver (as described in Section 5.1.3), except for the way it solves recurse constraints and resolution constraints. The rest of this section explains how recurse constraints and resolution constraints are solved.

5.3.1 Solve Recurse Constraints

To solve a recurse constraint $[\![p \wedge (s) : t]\!]$ we first create a random permutation of the rules where each permutation is equally likely to occur. We then instantiate and apply each rule until one succeeds or all rules have been tried. To instantiate a rule:

1. For every **new** s-constraint we associate to s a fresh term t, replace every occurrence of s by t, and drop the **new** s constraint.

2. Every reference (resp. declaration) is replaced by an *occurrence*. Occurrences has a position in addition to a name such that two occurrences with different positions are distinct. Identical occurrences are assigned the same position.

3. Every variable is replaced by a variable with a fresh name.

Step (1) is necessary to distinguish variables from ground terms in an NaBL2 specification. We have ignored **new** *s*-constraints until now, because the examples in this thesis use greek symbols or italic names to distinguish variables from ground terms. Textual NaBL2 specifications use **new** *s* to indicate that *s* is not a variable but a fresh, concrete term. Step (2) ensures that two distinct references (resp. declarations) remain distinct, even when they are assigned the same name later on. Step (3) ensures that the rule does not use a variable that is already being used in the program that is being generated.

To apply a rule all components (pattern, scope, and type) from the recurse constraint are unified with the components (pattern, scope, and type) from the rule. If any of these unifications fail, the rule cannot be applied. If all unifications succeed, the constraints from the rule are added to the program and the substitution is applied to the program.

Figure 5.8 shows the unification algorithm. An invocation $\mathcal{U}(t_1,t_2)$ of $\mathcal{U}$ on terms t_1 and t_2 yields a substitution σ if t_1 and t_2 can be unified and $\perp$ otherwise. The notation $\sigma(t)$ is used to represent the application of the substitution σ to the term t. The auxiliary function vars(t) collects all variables in a term t and is used to prevent unifying a variable that contains the same variable (commonly referred to as the *occurs check*).

$$
\begin{aligned}
\mathcal{U}(x,x) &= \{\} \\
\mathcal{U}(t,x) &= \mathcal{U}(x,t) \\
\mathcal{U}(x,t) &= \begin{cases} \{x \mapsto t\} & \text{if } x \notin \text{vars}(t) \\ \perp & \text{otherwise} \end{cases} \\
\mathcal{U}(c(t_1,\ldots,t_n),c(t'_1,\ldots,t'_n)) &= \mathcal{U}([t_1,\ldots,t_n],[t'_1,\ldots,t'_n]) \\
\mathcal{U}([x \mid xs],[y \mid ys]) &= \mathcal{U}(\sigma(xs),\sigma(ys)) \cup \sigma \text{ where } \sigma = \mathcal{U}(x,y) \text{ and } \sigma \neq \perp \\
\mathcal{U}(_,_) &= \perp \text{ if none of the above match}
\end{aligned}
$$

Figure 5.8: Unification algorithm

5.3.2 Solve Resolution Constraints

While generating a program we do not assign a concrete name to references and declarations. Instead, we leave the name of references and declarations variable until the program is made concrete (see Section 5.3.6). As a consequence, the name of a reference (resp. declaration) in the scope graph can be variable. Concrete names and variable names affect the visibility during name resolution as follows:

- When resolving a concrete name, a declaration of the same name hides more distant declarations (both concrete and variable).
- When resolving a variable name, a declaration of a concrete name hides more distant declarations of the same name.

To solve a resolve constraint $x_i^{\mathsf{R}} \mapsto \delta$ we first compute all declarations d_j^{D} that are visible from x_i^{R}. We create a random permutation of the declarations (where each permutation is equally likely to occur) to prevent a reference from always resolving to a deterministic declarations (such as the closest declaration). Then, for each declaration d_j^{D}, we unify δ with d_j^{D} (to solve the resolve constraint) and unify x with d (to encode the fact that the reference and declaration should have the same name). If the generator needs to backtrack it will first consider different solutions to a constraint which corresponds to resolving the reference to a different declaration.

Note that solving a resolve constraint this way does not guarantee that the reference will indeed resolve to the declaration in the final program. First, since references are being resolved while the scope graph is not complete, it is possible that new declarations are added along the way. Second, closer declarations that have a variable name may hide declarations that are more distant if the closer declaration with a variable name is assigned the same name as the more distant declaration. For these reasons we need to check afterwards whether every reference indeed resolves to the intended declaration (see Section 5.3.4).

5.3.3 Consistency Check

The consistency check prevents the generator from continuing generation with a program for which it is obvious that it can never become well-formed. In particular, we check the following properties and backtrack when either property does not hold:

- For every typing constraint $x_i^{\mathsf{D}} : t_1$ we check that there is no entry $x_i^{\mathsf{D}} : t_2$ in the type environment such that $\mathbb{U}(t_1, t_2) = \bot$, i.e. t_1 and t_2 cannot be unified.
- For every equality constraint $t_1 \equiv t_2$ we that check t_1 can unify with t_2.

5.3.4 Resolution Check

As explained in Section 5.3.2, the generator solves a resolution constraint by resolving a reference to a random visible declaration. However, the scope graph may change during generation causing a declaration that was visible at the moment the reference was resolved to be hidden later. For example, new declarations may be added such that the declaration that the reference is supposed to resolve to is no longer visible. Similarly, the scope graph may contain scope variables. When a scope variable is replaced by a ground scope the structure of the scope graph changes causing certain declarations to no longer be visible. In both cases, the decision to resolve the reference to a certain declaration might become invalid later. To make sure the generator does not generate ill-formed programs we verify that for every intermediate program the decision to resolve a reference to the declaration

is still valid. This is done by keeping track of the resolutions and computing for every reference whether the declaration that it should resolve to is still visible. If the recorded declaration is no longer visible, then the decision to resolve the reference to the declaration has become invalid and the generator backtracks.

5.3.5 Algebraic Sorts

SDF provides special constructs to specify the repetition of a syntactic sort. Specifically, $A*$ denotes zero or more repetitions of the symbol A and $A+$ denotes one or more repetitions of the symbol A. However, the Spoofax-generated signature uses a single parametric sort `List(a)` to represent both lists of size zero or more and lists of size one or more. This can lead to the generation of terms that are syntactically invalid. For example, if the language requires a list of one or more statements it will fail to parse an empty list of statements.

To work around this issue our implementation derives a custom signature based on the SDF3 files. This custom signature uses the sorts `IterStar(a)` and `Iter(a)` to distinguish lists of size zero or more and lists of size one or more, respectively. Constructors for these sorts are implicitly added to the signature (see Figure 5.9).

```
// List
Nil     : IterStar(a)
Cons    : a * IterStar(a) -> Iter(a)
Iter(a) : IterStar(a)

// Optional
Some    : a -> Option(a)
None    : Option(a)
```

Figure 5.9: Constructors and injections that are implicitly added to a signature.

Stratego signatures are order-sorted: a term of sort s_1 can be used when a term of sort s_2 is provided if s_1 is a subsort of s_2. When solving a recurse constraint we may only use rules that produce a term of the correct syntactic sort. For this reason we annotate every NaBL2 rule and every recurse constraint with a sort. When solving a recurse constraint we check that the sort of the rule that is applied is a subsort of the sort annotated to the recurse constraint.

Moreover, as the definition of `Iter(a)`, `IterStar(a)`, and `Option(a)` shows, sorts can be polymorphic. When solving a recurse constraint, the sort of the rule that is being applied needs to be a subsort of the sort of the recurse constraint. To ensure this is the case we compute all subsorts of the sort of the recurse constraint and test whether any of these sorts unify with the sort of the rule. For example, when the recurse constraint requires a term of sort `IterStar(Declaration)`, a rule that produces the term `Cons(_, _)` of sort `Iter(a)` can be used because `Iter(a)` is a subsort of `IterStar(a)` for any `a`.

5.3.6 Concretize

Once all constraints are solved the program is converted to its textual form. First, every reference–declaration pair with a variable name is assigned a fresh name. Any remaining variables in the AST are replaced by a value based on their sort. For example, a declaration that no reference resolves to is represented by a variable in the AST. Similarly, the constraint generation rules typically leave literal values (such as integer literals) variable. The sort of a variable is computed based on the position of the variable in the term. If the sort is `INT` the variable is replaced by a random integer between 0 and 100. If the sort is `ID` the variable is replaced by 'r' followed by a an increasing integer. Finally, the term is pretty-printed to get a textual representation of the program. To pretty-print the term we first convert it to a Spoofax-compatible representation and then invoke the Spoofax-generated pretty-printer.

5.4 Term Generation Example

We demonstrate the generation algorithm using `L1`, the language that was introduced in Section 5.1. The syntax and static semantics for `L1` are described in Figure 5.6 and Figure 5.7, respectively. Throughout the generation process, the generator keeps track of the partial AST (pattern), type environment (types), intended name resolutions (resolutions), and remaining constraints (constraints). The generator may generate a well-typed term as follows:

1. The generator starts with a variable pattern and a single recurse constraint:

Pattern	**Resolutions**
e	$\varnothing$
Types	**Constraints**
$\varnothing$	$[\![e \wedge (\mathrm{s}) : t]\!]$

2. The generator picks a random constraint. Since there is only one constraint, it must pick the recurse constraint. The generator solves this constraint by applying a random constraint derivation rule; assume it applies the addition rule (rule 5.3).

Pattern	**Resolutions**
$e_1 + e_2$	$\varnothing$
	Constraints
Types	$[\![e_1 \wedge (\mathrm{s}) : \mathrm{Int}]\!]$
$\varnothing$	$[\![e_2 \wedge (\mathrm{s}) : \mathrm{Int}]\!]$

3. The generator again picks a random constraint; assume it picks the recurse constraint for pattern e_2. The generator solves this constraint by applying a random constraint derivation rule; assume it applies the variable rule (rule 5.2).

Pattern	**Resolutions**
$e_1 + x_i$	$\varnothing$
	Constraints
Types	$[\![e_1 \wedge (s) : \text{Int}]\!]$
$\varnothing$	$x_i^{\mathsf{R}} \longrightarrow s$, $x_i^{\mathsf{R}} \mapsto \delta, \delta : \text{Int}$

4. The generator again picks a random constraint; assume it picks the recurse constraint for pattern e_1. The generator solves this constraint by applying a random constraint derivation rule; assume it applies the integer literal rule (rule 5.1).

Pattern	**Resolutions**
$n + x_i$	$\varnothing$
Types	**Constraints**
$\varnothing$	$x_i^{\mathsf{R}} \longrightarrow s$, $x_i^{\mathsf{R}} \mapsto \delta, \delta : \text{Int}$

5. There are two constraint left that needs solving: $x_i^{\mathsf{R}} \mapsto \delta$ and $\delta : \text{Int}$. The former cannot be solved because there are no declarations that x_i^{R} can resolve to. The latter cannot be solved because δ is non-ground. The generator now backtracks, making different choices, until it finally reaches the state in step 2. The generator may then proceed as follows:

Pattern	**Constraints**
$e_1 + e_2(e_3)$	$[\![e_1 \wedge (s) : \text{Int}]\!]$
Types	$[\![e_2 \wedge (s) : \text{Fun}(t, \text{Int})]\!]$
$\varnothing$	$[\![e_3 \wedge (s) : t]\!]$
Resolutions	
$\varnothing$	

6. The generator again picks a random constraint; assume it picks the recurse constraint for pattern e_1. The generator solves this constraint by applying a random constraint derivation rule; assume it applies the integer literal rule (rule 5.1).

Pattern	**Resolutions**
$n_1 + e_2(e_3)$	$\varnothing$
	Constraints
Types	$[\![e_2 \wedge (s) : \text{Fun}(t, \text{Int})]\!]$
$\varnothing$	$[\![e_3 \wedge (s) : t]\!]$

7. The generator again picks a random constraint; assume it picks the recurse constraint for pattern e_3. The generator solves this constraint by applying a random constraint derivation rule; assume it applies the integer literal rule (rule 5.1). This refines the type of e_2 from $\text{Fun}(t, \text{Int})$ to $\text{Fun}(\text{Int}, \text{Int})$.

Pattern	**Resolutions**
$n_1 + e_2(n_2)$	$\varnothing$
Types	**Constraints**
$\varnothing$	$[\![e_2 \wedge (\mathrm{s}) : \mathrm{Fun}(\mathrm{Int}, \mathrm{Int})]\!]$

8. The generator again picks a random constraint. Since there is only one constraint left, it picks the recurse constraint for pattern e_2. The generator solves this constraint by applying a random constraint derivation rule; assume it applies the function abstraction rule (rule 5.5).

Pattern	**Resolutions**
$n_1 + \mathbf{fun}(x_i : t)\{e\}(n_2)$	$\varnothing$
Types	**Constraints**
$\varnothing$	$[\![t \wedge (\mathrm{s}) : \mathrm{Int}]\!], [\![e \wedge (\mathrm{s'}) : \mathrm{Int}]\!],$ $\mathrm{s'} \longrightarrow \mathrm{s}\ ,\ x_i^{\mathsf{D}} \longleftarrow \mathrm{s'}\ , x_i^{\mathsf{D}} : \mathrm{Int}$

9. The generator again picks a random constraint; assume it picks the recurse constraint for pattern e. The generator solves this constraint by applying a random constraint derivation rule; assume it applies the variable reference rule (rule 5.2).

Pattern	**Constraints**
$n_1 + \mathbf{fun}(x_i : t)\{y_j\}(n_2)$	$[\![t \wedge (\mathrm{s}) : \mathrm{Int}]\!],$
Types	$\mathrm{s'} \longrightarrow \mathrm{s}\ ,\ x_i^{\mathsf{D}} \longleftarrow \mathrm{s'}\ ,$
$\varnothing$	$x_i^{\mathsf{D}} : \mathrm{Int},\ y_j^{\mathsf{R}} \longrightarrow \mathrm{s'}\ , y_j^{\mathsf{R}} \mapsto \delta, \delta : \mathrm{Int}$
Resolutions	
$\varnothing$	

10. The generator again picks a random constraint; assume it picks the recurse constraint for pattern t. The generator solves this constraint by applying a random constraint derivation rule; assume it applies the integer type rule (rule 5.7).

Pattern	**Resolutions**
$n_1 + \mathbf{fun}(x_i : \mathbf{Int})\{y_j\}(n_2)$	$\varnothing$
Types	**Constraints**
$\varnothing$	$\mathrm{s'} \longrightarrow \mathrm{s}\ ,\ x_i^{\mathsf{D}} \longleftarrow \mathrm{s'}\ ,$ $x_i^{\mathsf{D}} : \mathrm{Int},\ y_j^{\mathsf{R}} \longrightarrow \mathrm{s'}\ , y_j^{\mathsf{R}} \mapsto \delta, \delta : \mathrm{Int}$

11. The generator again picks a random constraint; assume it picks the resolve constraint $y_j^{\mathsf{R}} \mapsto \delta$. The generator solves this constraint by resolving the reference to a random visible declaration. Since the only visible declaration is x_i, the generator resolves y_j to x_i. This replaces δ by x_i (to solve the constraint) and y by x (to encode that the reference and declaration have the same name).

Pattern	**Resolutions**
$n_1 + \mathbf{fun}(x_i : \mathbf{Int})\{x_j\}(n_2)$	$x_j \mapsto x_i$
	Constraints
Types	s' ⟶ s , x_i^{D} ⟵ s' ,
$\varnothing$	x_i^{D} : Int, x_j^{R} ⟶ s'

12. The generator again picks a random constraint. The generator picks the only constraint that needs solving; the type declaration constraint for x_i^{D}. The generator solves this constraint by recording that x_i^{D} has type **Int**.

Pattern	**Resolutions**
$n_1 + \mathbf{fun}(x_i : \mathbf{Int})\{x_j\}(n_2)$	$y_j \mapsto x_i$
Types	**Constraints**
x_i^{D} : Int	s' ⟶ s , x_i^{D} ⟵ s' , x_j^{R} ⟶ s'

The final program has no constraints that need solving and the generator returns the final program. Any remaining variables are assigned a value based on their sort. In this example, first x_i and x_j are replaced by a fresh name. Next, both n_1 and n_2 are replaced by a random integer in the range $[0, 100)$. Finally, the AST is converted to a representation that is understood by Spoofax and the Spoofax-generated pretty-printer is invoked on this AST to get a textual representation of the program.

5.5 Discussion

In this section we discuss the term generation algorithm. We first look at whether the generator guarantees well-formedness of the intermediate programs. We continue with a discussion of the correctness of the generation algorithm. Finally, we describe several choices that were made in the design of the algorithm.

5.5.1 Well-Formedness of Intermediate Programs

Roughly speaking, the generator creates a well-formed program by repeatedly solving one of its constraints. A natural question is whether the generator guarantees that before and after solving a constraint the partial program is well-formed. To answer this question, we first need to make precise what it means for a partial program to be well-formed. For the purpose of generating well-formed programs, the obvious definition of a well-formed partial program is a partial program that can be completed in a way that it becomes well-formed. However, this problem is equivalent (or even subsumes) the type inhabitation problem: in a given type environment, does there exist a term with a given type? Under the assumption that type inhabitation is undecidable for arbitrary NaBL2 specifications there does not exist an algorithm that decides whether a partial program is well-typed.

From this discussion it is clear that at best we can only guarantee a weaker form of well-formedness. Consequently, there must be partial programs that satisfy this weaker

form of well-formedness but that fail to satisfy this stronger form of well-formedness. If the generator generates such a partial program it essentially gets 'stuck': there is no way to complete the program such that it becomes well-formed. In fact, the only way for the generator to recover from such a stuck state is by backtracking. Since backtracking is expensive we would like to avoid getting into such a stuck state and this is what motivated the 'consistency check' in the generation algorithm. The consistency check rejects partial programs for which it is easy to determine that the generator is stuck.

For example, after resolving a reference to a declaration the typing constraints may contradict each other (i.e. assign two different types to the same name). It is obvious that no matter how the generator continues, the partial program will never become well-formed. By checking, for every intermediate program, if there is a contradiction in the typing constraints, the generator can backtrack as soon as it recognizes such a program. However, there are also more subtle ways to get into a stuck state that we do not check. For example, in L2, if the generator creates an assignment $lhs = e$, then it needs to generate subterms lhs and e. Syntactically, the only valid choices for lhs are a variable reference x or a field reference $e\,.\,x$. If the program does not contain any variable declarations or record definitions, a variable reference or field reference will never resolve. That means that the generator got stuck the moment it created the assignment $lhs = e$.

Of course, we could extend the consistency check to cover this case as well. The question then becomes how extensive the consistency check should be. If we keep extending the consistency check, then at some point the consistency check becomes a search algorithm for well-formed terms on its own and the cost of computing consistency of a partial program would outweigh its benefit.

5.5.2 Correctness of the Generation Algorithm

A generation algorithm is correct if it is both sound and complete. A sound generator generates only well-formed programs; a complete generator generates every well-formed program with a non-zero probability. Together these two properties guarantee that a correct generator only generates well-formed programs and that no well-formed programs are structurally excluded.

The generation algorithm described in this chapter satisfies both properties. To see why the generator is sound, note that a program is well-formed if and only if all constraints can be solved. The generator only outputs a program if all constraints are solved. Constraints are solved under the same conditions as the original NaBL2 constraint solver, which has been proven to be sound [3]. The only exception is name resolution in an incomplete scope graph: we may optimistically resolve a reference to a visible declaration, but since the scope graph is incomplete, the declaration may become invisible later. This is why we keep track of to the declaration a reference is supposed to resolve to and check at every step whether this is still possible (as described in Section 5.3.4). Ergo, every generated program is well-formed.

To see why the generator is complete, note that the generator repeatedly solves a random constraint. Hence, there is a non-zero probability that all recurse constraints are solved before any other constraint is solved. A recurse constraint is solved by applying a random

constraint generation rule. Together, this means that the generator can derive any AST before solving any of the other constraints. This is precisely what would happen if we would generate a random syntactically valid AST first and compute the analysis results second.

5.5.3 Design Choices

While designing the generation algorithm we made several choices that have been left implicit. The rest of this section make these choices explicit and motivates these choices. A more in-depth discussion with suggestions for alternative choices can be found in Chapter 8.

Top-down generation The algorithm described in this chapter generates terms in a top-down order, starting at the root and repeatedly expanding a random leaf. However, there is no fundamental reason to generate terms in a top-down order. In fact, in some cases a top-down generation algorithm may not be ideal. Consider, for example, field references such as those found in Java. A field reference $e \,.\, x$ consists of two subterms: the name of the field x and a receiver expression e that determines in which scope the field reference should be resolved. The typical way to model this in NaBL2 is to create a reference with the name of the field x and to resolve this reference in a scope that is determined by the receiver expression e:

$$[\![e \,.\, x \wedge (s) : t]\!] := [\![e \wedge (s) : \text{TRec}(d_1)]\!],\ d_1 \rightarrowtriangle cs\ ,\ x_i^{\mathsf{R}} \longrightarrow \text{s'}\ , x_i^{\mathsf{R}} \mapsto \delta_2, \delta_2 : t,\ \text{s'} \longrightarrow cs\ .$$

However, the receiver expression has yet to be generated. To generate a receiver expression of the correct type the generator may again create a field reference such that it now needs to satisfy $(e \,.\, y) \,.\, x$. This leads to an increasingly complex term for which it becomes less likely that the generator succeeds at generating a well-formed term. If, on the other hand, the receiver expression were generated first, then the scope in which the reference should be resolved is known. This could make it easier to determine whether the reference can resolve to a declaration.

Generate full term While generating a well-typed term, the generator carries around the full generation context (e.g. partial term, type environment, established resolutions, constraints). In particular, the algorithm does not generate any part of the term in isolation. This is motivated by the observation that, in general, constraints that are derived in one branch of the term may interact with constraints that are derived in another branch. In practice, however, many programming language constructs are independent of each other or interact with each other in a very specific way.

Consider, for example, the lambda calculus. A term in the lambda calculus is either a variable reference, a function abstraction, or a function application. From the perspective of well-formedness, there are two kinds of dependencies between terms. First, the type environment is always passed downward such that sibling terms cannot modify each other's type environment. Second, a function application creates a dependency between the type of

the function ($\tau_1 \rightarrow \tau_2$) and the type of the argument (τ_1). When generating terms in a top-down order, it is possible to generate sibling terms one after another without violating these dependencies. By generating one term after another instead of generating terms simultaneously we can reduce the search space, which in turn reduces the cost of backtracking.

Expand program with random rule The generator solves a recurse constraint by applying a random constraint derivation rule. In particular, each constraint derivation rule is equally likely to be used to solve the recurse constraint. However, certain rules are more likely to make the generator diverge whereas other rules are more likely to make the generator converge. For example, to generate an term with an integer type the generator can choose to create an addition $e_1 + e_2$ or an integer literal n. The former creates two new recurse constraints, whereas the latter creates no new recurse constraints. Hence, the former rule is more likely to make the generator diverge (until the size bound forces the generator to backtrack), whereas the latter is more likely to make the generator converge (perhaps never even reaching the size bound). We suspect that by carefully choosing how to expand the program we can increase the likelihood that the generator succeeds at generating a complete program.

Solve random constraint The algorithm repeatedly solves a random constraint and backtracks when a constraint cannot be solved. The decision to solve a *random* constraint is motivated by two observations. First, it may not be possible to solve a constraint the moment it is encountered. For example, solving a resolution constraint depends on the existence of a visible declaration. If the syntactic structure of a programming language requires references to be created before a declaration, then it may not be possible to resolve the reference until we also generate the declaration. In that case the generation algorithm would fail to generate any term. Second, by solving a random constraint there is a non-zero probability that the generator solves all recurse constraints before any other constraint. This ensures completeness of the generator, i.e. every well-typed term can be generated and no term is structurally excluded. Many variations of this strategy are possible, and it would be interesting to investigate if solving constraints in a different order could improve the performance of the generator.

5.6 Related Work

The work in this chapter is closely related to the work of Pałka et al. [47] and Fetscher et al [19]. Pałka et al. describe a technique to generate random well-typed lambda terms based on the specification of the type system for the simply-typed lambda calculus. The type system for the lambda calculus is specified as a set of inference rules which suggests a straightforward generation procedure: to generate a term that is in the consequence of a rule, it is firstly necessary to generate terms that are in its premises. By repeating this procedure until all premises are satisfied, a well-typed lambda term is generated.

Fetscher et al. generalize this idea by generating well-typed terms in an arbitrary language whose type system is specified in Redex. A type system in Redex consists of a set

of inference rules that contain typing judgments. To perform arbitrary computations an inference rule may invoke a *metafunction*. For example, a metafunction may be used to look up the type of a name in a type environment. Metafunctions consist of a list of clauses that match a pattern and produce a term (similar to a *rewrite rule* in Stratego). The list of clauses is ordered: a term is only matched against the pattern of some clause if the term did not match the pattern of any of the previous clauses. The metafunctions are converted to inference rules, but to account for the ordering, equality- and disequality constraints are added as premises to the inference rule. When generating a program the generator now needs to make sure that all equality- and disequality constraints are satisfied.

Unlike Pałka et al. (but similar to Fetscher et al.) the algorithm described in this chapter is language-parametric, i.e. not tied to the lambda calculus. Terms in the lambda calculus have few dependencies and the generator described by Pałka et al. exploits these dependencies. First, in the lambda calculus the typing context is always passed 'downward' such that declarations that are added in one branch of the term are not visible in another branch. Since subterms do not influence each other, each subterm can be generated in isolation. Second, the order in which you generate subterms does not influence well-typedness. For example, to generate an application-term, it does not matter whether you generate the function first and the argument second or vice versa. These two observations allow a generation procedure in which subterms are generated independently and one after another. These properties do not hold for arbitrary languages defined in NaBL2, which is why our algorithm is more general, but also more expensive in terms of runtime cost.

Compared to Fetscher et al. the algorithm described here operates on an NaBL2 specification instead of a Redex specification. NaBL2 and Redex take different approaches with respect to specifying a language's static semantics. First, NaBL2 is a relatively rich language that models name resolution and typing, among other concepts. Redex is a relatively simple language that models inference rules and typing judgments and uses metafunctions to define arbitrary computations. Second, Redex is a lot more restrictive. In Redex, a type system is defined as a set of inference rules, where a typing judgment consists of input terms (such as a type environment and an expression to determine a type for) and output terms (such as the computed type). As a consequence, the type of a term (output) is defined in terms of the type environment and the term (inputs). The inference rules explicitly pass around the context (Γ) and the typing judgments must be satisfied with respect to this context. None of these restrictions apply to NaBL2. In particular, the constraint derivation function disconnects constraints from the AST and the satisfiability of constraints is determined with respect to the set of *all* constraints (notably, this set is incomplete during generation).

Besides the fact that NaBL2 and Redex are two different specification languages, there are two fundamental differences in the generation algorithm. First, the algorithm described by Fetscher et al. keeps track of a stack of goals (premises in the inference rules) that need to be satisfied. This suggests that programs are generated depth-first, which indicates that the generator is unable to deal with mutually recursive dependencies. Second, the algorithm described by Fetscher et al. converts metafunctions to inference rules, essentially making the metafunctions part of the object language. For example, name resolution needs to be encoded as a metafunction and is then compiled to an inference rule. NaBL2, on

the other hand, has a built-in notion of name resolution, which allows us to optimize the implementation.

Chapter 6

Term Generator Evaluation

We evaluate the effectiveness of the term generator at discovering compiler bugs by performing two experiments. In the first experiment we measure the effectiveness of the term generator at discovering bugs in 35 MiniJava [4] compilers that were developed by students at Delft University of Technology. In the second experiment we measure the effectiveness of the term generator at finding type soundness bugs in L1, L2, and L3 [49], three simple programming languages of increasing complexity. We repeat the same experiment on Tiger [4] to investigate how our approach scales to languages that are more complex.

6.1 Conformance Testing

As part of the Compiler Construction course at Delft University of Technology, students are provided a compiler frontend (parser and static analyzer) for MiniJava [4] and tasked with implementing the compiler backend. Student submissions are graded using a handcrafted test suite that has been developed and improved over the past several years. The test suite consists of valid MiniJava programs and the expected result of their execution.

6.1.1 Setup

To evaluate the effectiveness of our term generator at discovering compiler bugs, we investigate whether we can use the generated terms to identify erroneous compilers (according to the handcrafted test suite). Specifically, we generated 5,000 well-typed MiniJava terms and evaluated these terms using each of the 35 compilers from the 2015-2016 edition of the Compiler Construction course. Since the correct outcome for a given test input is not known upfront (also known as the *oracle problem*), we use the majority outcome as an estimator for the correct outcome. In addition, we compare the majority outcome to the outcome on a reference implementation that has been developed by the lecturer. Since the generated programs are not guaranteed to terminate we kill a program if it does not terminate within 4 seconds. For each program, we registered the standard output and standard error as well as whether the program terminated or was killed after the timeout.

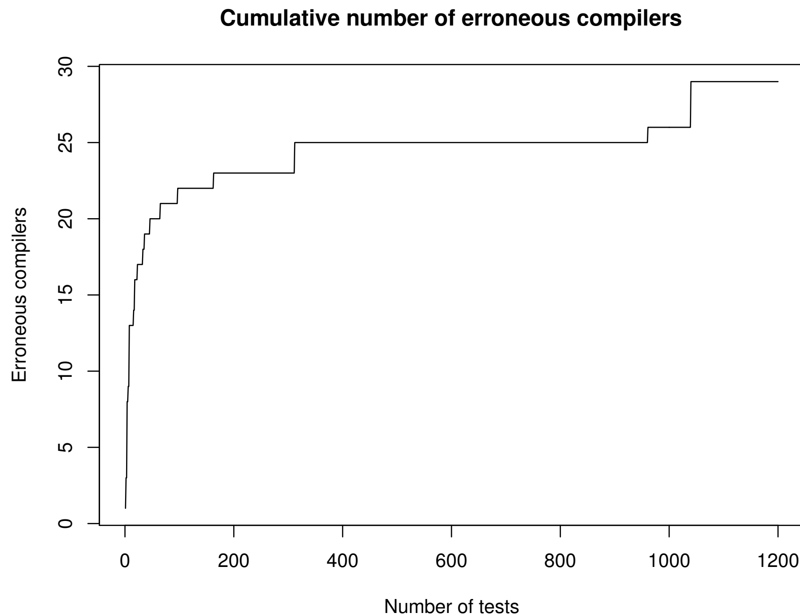


Figure 6.1: Cumulative number of detected erroneous MiniJava compilers as a function of the number of executed tests.

6.1.2 Results

Figure 6.1 shows the cumulative number of detected erroneous compilers as a function of the number of executed tests. The graph shows that the first 46 tests detect 20 erroneous compilers and that after 1040 tests, no new erroneous compilers are detected (for brevity, we have omitted tests 1,201–5,000 from the plot). This shows that very few tests were sufficient to detect bugs in the majority of compilers and the advantage of generating more tests quickly declines. In total 29 erroneous compilers were detected. From these 29 compilers, 28 compilers failed one or more tests in the handcrafted test suite. Surprisingly, however, is that the generated test suite caught one compiler that the handcrafted test suite missed, and that the generated test suite missed one compiler that the handcrafted test suite caught.

The compiler that was caught by the handcrafted test suite but missed by the generated test suite contains a bug causing calls to methods in ancestor classes to fail. Nothing prevents the generator from generating a program that exposes this bug. We suspect that the probability of generating such a term is simply too small for it to be present in a sample of 5,000 terms.

The test that identified a bug in a compiler that was presumed to be correct is shown in Figure 6.2. To understand why the compiler failed this test, we need to look at the Java Virtual Machine (JVM) specification. Specifically, the JVM does not have a representation for Boolean values. Instead, Boolean values are represented as integers with ‘false’ being defined as the integer ‘0’ and ‘true’ as any non-zero integer. Additionally, when creating a new object the JVM initializes fields to a default value based on the type of the field. For Boolean fields the default value is ‘false’, which is represented by the JVM as the integer

'0'. Based on these semantics the correct behavior for the program in Figure 6.2 is to loop indefinitely. However, this particular compiler represented 'true' as '0' and 'false' as '-1'. As a consequence, instead of looping indefinitely, the compiled program terminates.

Note that this mistake can easily go undetected: if the compiler consistently represents 'true' as '0' and 'false' as '-1' a program involving Boolean values will produce the correct behavior. The bug only becomes visible when the compiled code interacts with a system that represents Boolean values differently such as the JVM or binaries that were compiled with a different (correct) compiler. We believe that it is unlikely that a human would test for this bug and even less likely that someone would create a test case that relies on the default behavior of the JVM to trigger the bug. Indeed, the handcrafted test suite that was used in the past two editions of the course did not contain a test that exposes this bug even though multiple graduate students have been involved with its creation.

```
class E {
  public static void main (String[] w) {
    while(new n3().n2()) {
    }
  }
}

class n3 {
  boolean n1;

  public boolean n2() {
    return !n1;
  }
}
```

Figure 6.2: Test case that exposes a bug in a compiler that was presumed to be correct.

6.2 Type Soundness Testing

How effective is the generator at discovering type soundness bugs? To answer this question we created six mutations of L1 that contain a type soundness bug. The mutants are created by making the type system more liberal (accepting previously ill-typed terms as well-typed terms) or by making the interpreter more restrictive (failing to interpret well-typed terms that it could previously interpret). Details on how the mutants were created are shown in Appendix B. For each mutant, we measured the time it takes the generator to generate a term that "goes wrong", i.e. that triggers the type soundness bug. In mutation testing terminology, we say that the test "kills the mutant".

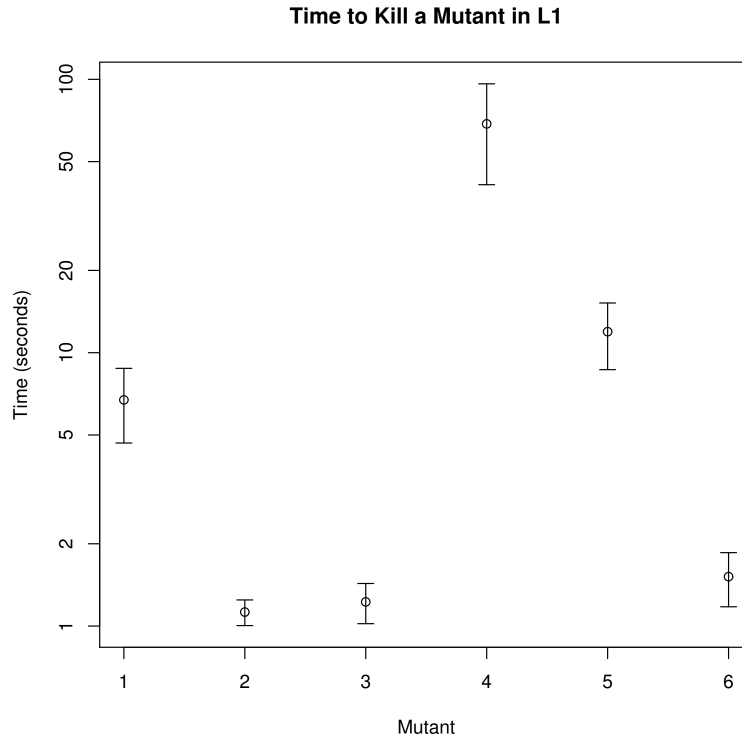


Figure 6.3: Mean time to kill mutants 1 to 6 with a 95% confidence interval.

6.2.1 Results

The mean time to kill each of the six mutants as well as a 95% confidence interval is shown in Figure 6.3. The figure shows that all mutants can be killed within reasonable time. It takes the least time to kill mutant 2 (averaging 1.1 seconds) and the most time to kill mutant 4 (averaging 69 seconds).

Based on these results, we wonder how the complexity of the language under test influence the effectiveness of the generator at discovering type soundness bugs. To answer this question we repeat the previous evaluation on L1, L2, and L3, three languages of increasing complexity. L2 extends L1 by introducing records and L3 extends L2 by replacing records by classes with inheritance and adds null-values to the language. We created six mutants for each language by injecting the same error as in the preceding evaluation and measured the time it took to kill the mutant.

The results are shown in Figure 6.4. As expected, killing a mutant becomes increasingly difficult as the language becomes more complex. However, the results also make precise how much more difficult it becomes. Killing a mutant in L2 compared to L1 takes anywhere between 2.5 (M3) and 94 (M4) times as much time on average. Analogously, killing a mutant in L3 compared to L2 takes anywhere between 2.0 (M5) and 5.1 (M4) times as much time on average. The hardest mutant to catch was M4 in L3, taking on average more than nine hours.

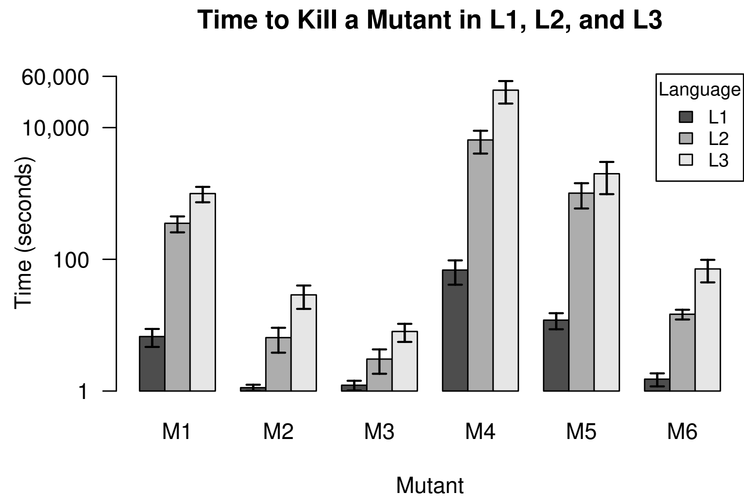


Figure 6.4: Mean time to kill mutants 1 to 6 in the languages L1, L2, and L3 with a 95% confidence interval.

This makes us wonder how our technique scales to real-world programming languages. To answer this question we repeat the evaluation using Tiger [4], a language that is used for teaching Compiler Construction [1]. Tiger's static semantics are more complex than the static semantics of L1-L3 (it's NaBL2 definition is about five times as long), thereby providing a more realistic view of how the generator would perform on actual languages. We again created thirteen *mutants*: variations of the Tiger compiler that contain a type soundness bug. The mutants are based on what we think are typical programming errors. In fact, mutants 2, 8, 9, 12, and 13 are based on mistakes that we made ourselves while crafting the language specification and discovered during the evaluation. Appendix B lists the thirteen mutants and the bug that was introduced.

Figure 6.5 shows the average time it takes to kill the mutant together with a 95% confidence interval. For mutants 3, 6, 7, 9, 10, 11, and 13 we were unable to generate kill the mutant within 24 hours; these mutants have been omitted from the figure.

6.3 Threats to Validity

We identified several factors that might affect the causal relation suggested in our conclusion (internal validity):

- Tiger has not been proven to be type sound. If we assume Tiger is not type sound, then there exists well-typed terms that "go wrong" other than those introduced by the mutation.

[1] We used the Language Reference published by EPITA at `https://www.lrde.epita.fr/~tiger/tiger.html`

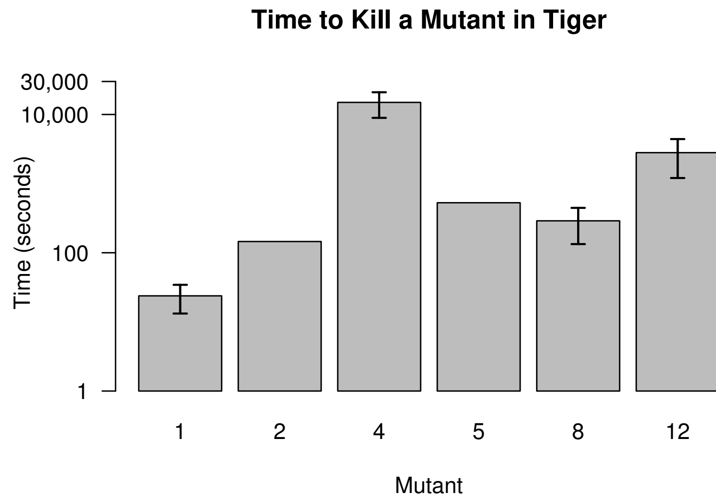


Figure 6.5: Mean time to kill mutant 1, 2, 4, 5, 8, and 12 with a 95% confidence interval.

- Our implementation of Tiger, notably its static and dynamic semantics, might contain a bug, causing the interpreter to "go wrong" on well-typed terms.
- Our generator might contain a bug, causing ill-typed terms to be generated and presented as well-typed terms.

We mitigate the first two threats by manually verifying that each reduction failure was caused by the error that we injected into the compiler. We mitigate the third threat by automatically verifying that the generated terms are indeed well-typed according to the Spoofax-generated static analyzer.

As to the generalizability of our results (the external validity), we identified the following threats:

- **Oracle Problem** In our evaluation involving MiniJava compilers, we use the majority outcome as an estimate of the correct outcome. There is a threat that the majority of compilers produces an identical but wrong answer, in which case we will incorrectly flag erroneous compilers as being correct and might incorrectly flag a correct compiler as being erroneous. We mitigate this threat by using a substantial number of compilers and by verifying that for every test the majority outcome agrees with the outcome of a reference compiler that we developed ourselves.
- **Profoundness of Bugs** In our study involving MiniJava compilers, we did not identify or categorize the bugs that caused a test to fail. In the worst case, we only found a single shallow bug. In the best case, every erroneous compiler contained a distinct bug, possibly even bugs that we had not thought of when developing the manual test suite but that are caught by the generated tests.
- **Termination Detection** We estimate non/termination by running a MiniJava or Tiger program for at most n seconds before killing it. If n is too short to let a program

terminate we incorrectly conclude that the program does not terminate. In MiniJava, this could incorrectly flag a compiler as erroneous (and vice versa). In Tiger, we could miss out on an opportunity to detect unsoundness (i.e. a false negative). We mitigate this threat by measuring the false negatives at $n+1$ compared to n and picking an n that gives an acceptable number of false negatives compared to the time it takes to run the evaluation.

- **Quality of Test Suite** We evaluated the generator by comparing a generated test suite to a manually written test suite. The results of this study should be interpreted relative to the quality of the manually written test suite. Specifically, if the manually written test suite is of low quality, it will be easier to discover new bugs using the generator. To mitigate this threat, we subjected the manually written test suite to the scrutiny of two researchers who have been closely involved with the course for multiple years.

Chapter 7

Analysis

Our evaluation on L1-L3 has shown that the effectiveness of the generator at discovering type soundness bugs decreases as the language becomes more complex. Moreover, our evaluation on Tiger has shown that many type soundness bugs cannot be detected at all within reasonable time. In this section we take a step back and try to explain why the generator is unable to discover these bugs.

7.1 Generator Throughput

A possible reason for the reduced effectiveness as the language becomes larger is the rate at which the generator generates terms. It is reasonable to assume that the less terms the generator generates, the longer it takes to discover a bug. Figure 7.1 shows the average number of terms that are generated within one minute for L1, L2, and L3. As can be seen, the complexity of the language significantly affects the throughput of the generator.

This makes us wonder whether the reduced performance is caused by the generator having a lower throughput, or the generated terms being less suitable for detecting soundness bugs. Figure 7.2 shows the same data as Figure 6.4 compensated for the decrease in throughput. Specifically, we scale the time it takes to discover the soundness bug in L2 and L3 as if the throughput would be equal to the throughput in L1.

In Figure 7.2, the time to kill a mutant no longer increases exponentially as the language becomes more complex. In fact, except for mutant 4, the time to kill a mutant in L3 is less than or comparable to the time it takes to kill the same mutant in L2.

This provides evidence in favor of the hypothesis that the effectiveness of the generator at discovering soundness bugs depends for a large part on the speed by which the terms are generated, and not so much on the complexity of the terms that are generated. In other words, if the language becomes more complex it takes more time to generate terms, but the terms that are generated are not less suitable for detecting soundness bugs.

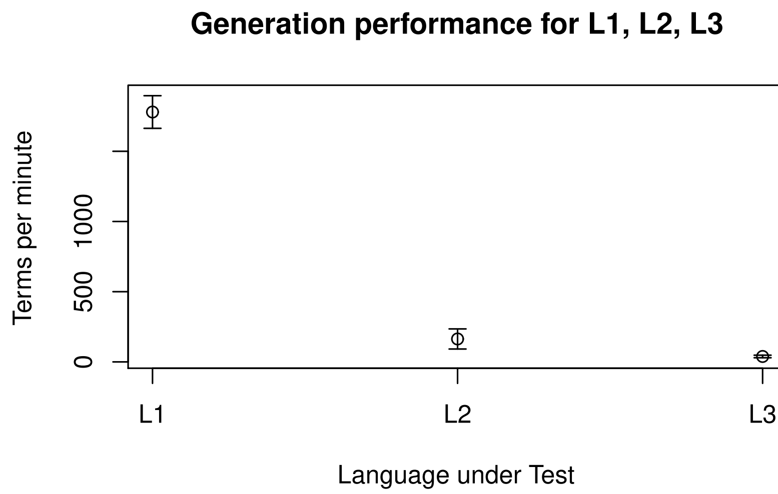


Figure 7.1: Average number of terms generated in one minute for L1, L2, and L3 with 95% confidence intervals.

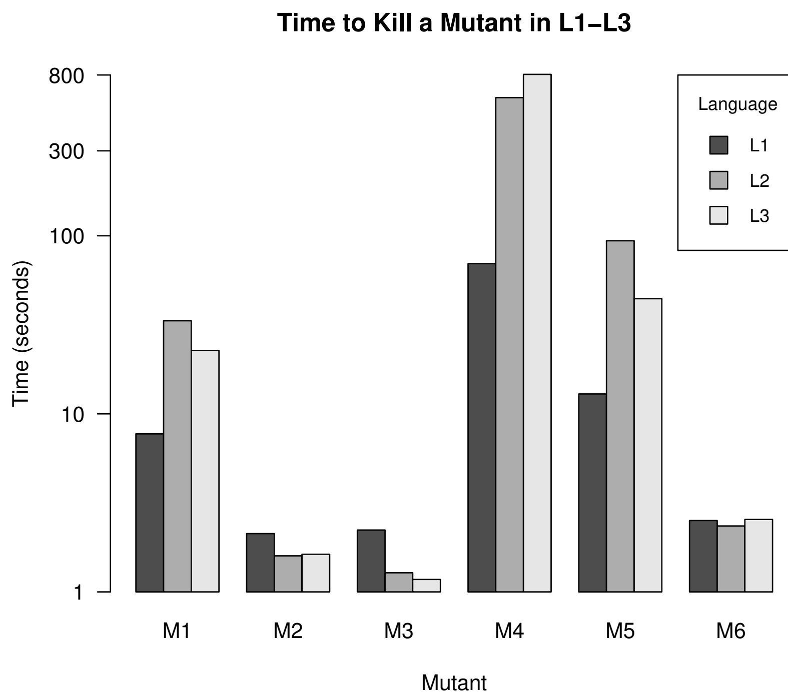


Figure 7.2: Mean time to kill mutant M1-M6 in languages L1, L2, and L3 compensated for generator throughput.

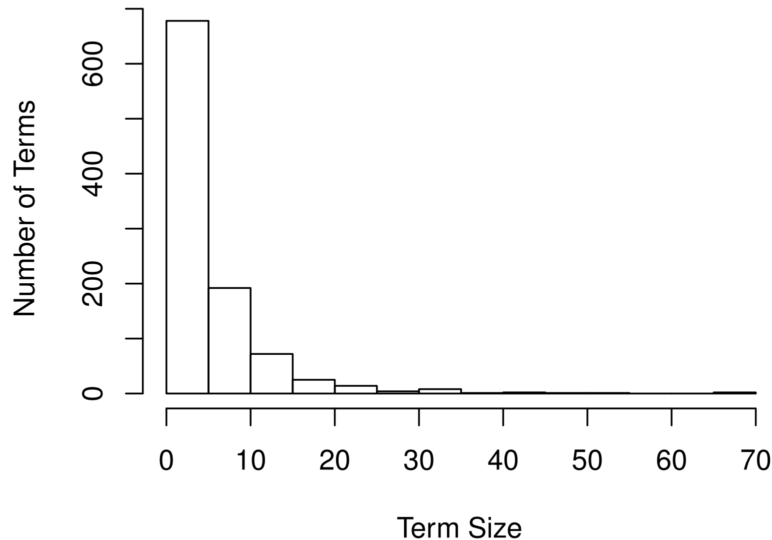


Figure 7.3: Histogram of term sizes for a random sample of 1,000 generated terms in L3.

7.2 Term Size

Smaller test cases are easier to communicate and to be understood by a developer. On the other hand, the rate of error detection is known to vary significantly as a function of the sizes of the test programs [61]. To get a sense of the distribution of the terms that are being generated, we compute the size of 1,000 terms for L3. We express the size as the number of constructor applications. That is, the size of a constructor application is one plus the sum of the size of its children. Figure 7.3 shows a histogram of the size of the generated programs. The histogram shows that the generator is heavily biased towards small programs (L1 and L2 show a similar distribution). Specifically, there are 678 terms of size 0-5, which are unlikely to expose any compiler bugs.

7.3 Number of Resolutions

Detecting certain mutants requires a program with a specific kind of binding structure. For example, mutant 4 can be detected by a program that evaluates a function that accesses a variable that it captured from its environment (i.e. the function acts as a closure). We suspect that few terms contain name resolution patterns, making it harder to kill this specific mutant. To quantify the number of resolutions, we generated 1,000 terms in L3 and measured the number of resolutions. Figure 7.4 shows a bar plot of the number of resolutions. It can be seen that there are 787 terms that have no name resolution, 106 terms where a single reference is resolved, etc. This shows that the generator is biased towards terms with little resolution.

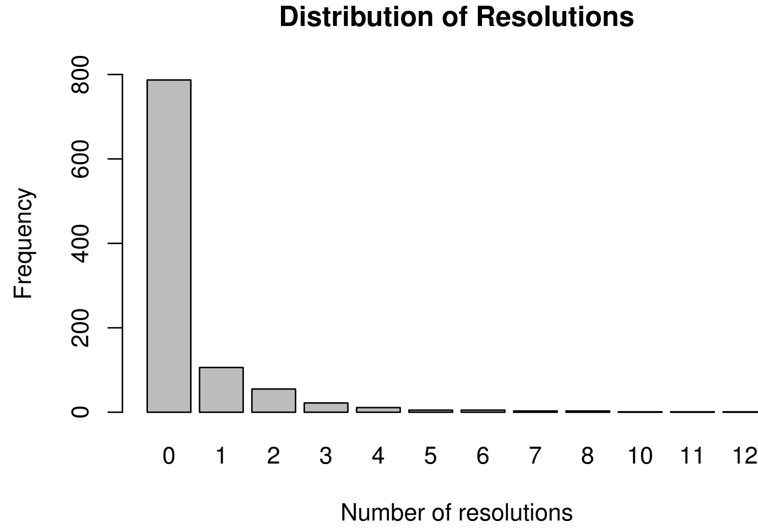


Figure 7.4: Distribution of number of resolutions for 1,000 generated terms in L3.

7.4 Term Redundancy

Every term is generated in isolation; the generator does not remember what terms it has previously generated. We suspect that this leads to some redundancy in the terms that are generated. To quantify the amount of redundancy we generated 1,000 terms in L1 and grouped terms that are *similar*, where we define two programs as similar if they are alpha-equivalent or if they differ only in their choice of literal values.

Table 7.1 shows the group sizes and the number of groups of this size. Specifically, there are 516 groups of size 1, 48 groups of size 2, 9 groups of size 3, etc. There is one group of size 46, which is the largest group. In total there are 599 groups, which means that $1000 - 599 = 401$ terms are similar to another term. This means that at least 40% of the generated terms are redundant. We suspect that by steering the generator towards different programs, its effectiveness at discovering compiler bugs can be greatly improved.

Group size	1	2	3	4	8	9	10	11
Frequency	516	48	9	3	1	1	1	5

Group size	12	13	14	16	17	18	19	20	46
Frequency	2	1	2	3	3	1	1	1	1

Table 7.1: Group sizes and the frequency with which the group size occurs after grouping similar terms from a set of 1,000 generated terms in L1.

Chapter 8

Discussion

In the preceding chapters we presented the term generation algorithm and evaluated its effectiveness at discovering different kinds of compiler bugs. In this chapter we reflect on these results and discuss some alternative choices in the design of a term generator that could improve its performance.

8.1 Algorithm Design Choices

Fundamentally, the term generation algorithm searches for well-typed terms by alternating between expanding the program and solving constraints, both of which involve making random choices. If the generator realizes that no such term can be found it backtracks and makes a different sequence of choices.

To make random testing of more complex languages possible this random search should become more efficient (generate more programs per second) as well as more effective (generate programs that are more likely to trigger certain bugs). One factor affecting efficiency is backtracking, which can be alleviated by either reducing the need for backtracking or reducing the cost of backtracking. What follows are several ideas of how we could improve this random search based on our observations and experiments. We suspect that a combination of some of these ideas (and other ideas) will be needed to make the generator performant enough to detect bugs in more complex languages.

Exploit dependencies The generator expands a partial program by repeatedly solving a random constraint and continuing generation with one of the resulting programs. When the generator cannot make any progress it backtracks. While backtracking the generator reverts the last choice and proceeds down a different path, choosing a different solution to the constraints and solving the constraints in a different order. The motivation to always consider the *complete* partial program is that there may be dependencies between different parts of the program, which prohibits generating these parts in isolation. In practice, however, there are few dependencies, and our approach leads to unnecessary backtracking. To see why this is the case, note that backtracking reverts *every* choice that was made, even in parts of the

program that are well-formed. Moreover, after backtracking the generator proceeds down a different path, but in many cases this path is equivalent with respect to well-formedness.

For example, consider a partial L1 program that consists of an expression $e_1 + e_2$, where e_i is a variable for which the program contains a recurse constraint. The generator will try to solve one of the recurse constraints before the other. While backtracking, the generator will consider solving the recurse constraints in a different order. However, the order in which you generate the operands of an addition in L1 does not affect well-formedness, so considering these constraints in a different order is useless. Next, imagine that at some point during generation, two out of three expressions are well-formed (when considered in isolation) and one is ill-formed. While backtracking the generator reverts the work that was done in each of the expressions, even though only one of the three expressions is ill-formed.

These observations suggest that by carefully analyzing the dependencies we can reduce the cost of backtracking. For example, if two parts of the program are independent, they can be generated in isolation and combined afterwards. If one part of the program depends on another part, then they can be generated in a fixed order (in particular, it is unnecessary to consider a different order). Perhaps it is possible to extract these dependencies from the NaBL2 specification, manually annotate the NaBL2 language with this information, or use a more declarative language that explicitly models these concepts. Either way, we suspect that reasoning about dependencies while generating terms could improve the performance of the generator.

Unreachable code Manual inspection of the generated programs revealed that many programs contain unreachable code, i.e. code that is never executed because there exists no control-flow path to the code from the rest of the program [15]. For example, a MiniJava program consists of a main class followed by one or more classes, each containing zero or more methods. A MiniJava program starts executing in the main method which consists of a single statement. Unless this statement creates an instance of a class and invokes a method on this instance, the code in the classes and methods is never executed. It can be argued that such programs are unlikely to uncover certain kinds of bugs, such as type soundness bugs, simply because the code is never evaluated. It would be interesting to explore the possibility of generating programs that contain less unreachable code and evaluate whether this makes the generated programs more effective at discovering type soundness bugs. For example, when incrementally expanding a program (as described earlier), the generator could consult the control-flow graph to avoid expanding unreachable code (since this is likely to be unreachable as well).

Bounding the size As described in Section 5.5, one of the parameters to the generator is the maximum size of the term it may generate (*maxSize*). The decision to impose such a size bound is motivated by the observation that without such a bound the generator is likely to diverge. However, for large size bounds the search space becomes large as well and the generator is likely to fail at generating a term.

An alternative approach is to generate a small well-formed program, introduce one or more placeholders, and search for a slightly larger well-formed program. For example, consider a language for arithmetic expressions. Given a well-formed program in this language

that contains an integer literal, we can replace the integer literal by the addition of that same integer literal and some unknown term (represented by a variable). The generator can then derive all constraints from this term (as if it were going to analyze the program). Among the constraints will be a recurse constraint that corresponds to the unknown term. Since the original program was well-formed most constraints should be trivial to solve such that the generator can focus on generating a concrete term for the unknown term.

The additional benefit of 'growing' a program like this is that it gives you more control over the size of the generated program. In the current approach the size bound acts solely as an upper bound, but for a large value of *maxSize* the generator may converge too fast, yielding only small terms. With the suggested approach you can keep growing a program until its size is within a certain range or exceeds a certain threshold. Preliminary experiments in `L1` showed that this strategy allows programs to be generated that are much larger than what is possible with the current algorithm.

Redundant programs In Section 7.4 we analyzed the redundancy in a set of 1,000 randomly generated `L1` terms by grouping terms that only differ in their choice for literal values. The results showed that 40% of the generated terms are similar, and hence do not contribute to discovering new bugs. This is not surprising, since terms are generated independent from each other and the generator always starts generating from the same empty program. Especially for simple languages such as `L1`, where the chance of generating a single integer is 25%, one can expect many similar terms. One way to alleviate this problem is to remember the choices that the generator makes while generating a term and to avoid making the same sequence of choices when generating subsequent terms. Alternatively, the generator could start generating from an earlier generated program by introducing a placeholder in the program, similar to how we described under *Bounding the size*.

Declarative meta-language The NaBL2 language for specifying static semantics is quite flexible. This makes NaBL2 very good at describing how to derive constraints from an AST, which was the goal of the NaBL2 language. However, this flexibility makes it more complicated to reason about the consistency of a partial program. For example, consider a language that supports `let`-expressions that consists of a list of declarations and an expression. Each declaration in a `let`-expression is only visible in the region following the declaration. The typical way to model this in NaBL2 is generate the outermost and innermost scopes for the `let`-pattern. Next, the list of declarations within the `let`-expression is processed and each declaration creates a new scope that is a child of the outermost scope. When processing the last declaration the last child scope is linked to the innermost scope to "close the chain". However, as long as the list of declarations is not fully processed, the innermost scope is disconnected from the other scopes. For analysis, this is not an issue, since all constraints are derived upfront. For generation this is an issue, because it is not obvious that the scopes will eventually connected unless special care is taken. We suspect that a more declarative meta-language, where patterns such as the one just described are made explicit, allow optimizations that are currently not possible.

Chapter 9

Related Work

The idea of generating sentences by rewriting a nonterminal to a sequence of terminals and nonterminals using a random production is not new and dates back to at least 1970, when Hanford described a 'syntax machine' for automatically generating syntactically correct programs for checking compiler frontends [23]. The sentence generation algorithm described in Chapter 3 is similar to the algorithm described by Hanford, except that our algorithm generates algebraic terms (as opposed to strings) from Kernel SDF (as opposed to BNF) which are then pretty-printed.

There exists several variations of this sentence generation algorithm that guarantee certain coverage criteria or a certain distribution of the sentences. Purdom's algorithm [50, 37, 38] generates a small set of short sentences from a context-free grammar such that each production of the grammar is used at least once (rule coverage). Lämmel [34] has shown that rule coverage is not sufficient for discovering practical bugs and proposes context-dependent branch coverage as an alternative coverage criterion. McKenzie [40] presented an algorithm for generating sentences of length *n* from a context-free grammar such that all strings of length *n* are equally likely. The algorithm described in Chapter 3 is not guided by a coverage criteria or a certain distribution. Instead, we generate sentences until an ambiguity is found and every production is equally likely to be chosen.

Work in the area of detecting ambiguities is mostly aimed at static checks and exhaustive search. This differs from the algorithm in Chapter 3 which performs a non-exhaustive random search. For example, Brabrand et al. [9] propose a technique for detecting ambiguities in a context-free grammar using a conservative approximation that statically analyzes the grammar. Basten and Vinju [5] filter productions from the grammar that certainly do not contribute to the ambiguity of the grammar followed by an exhaustive search on the reduced grammar. Moreover, Basten and Vinju [6] propose an evaluation method for locating causes of ambiguity in context-free grammars by automatic analysis of parse forests. The algorithm in Chapter 3 does not make an attempt to locate the cause of ambiguities. Instead, it automatically shrinks the ambiguous sentence to a smaller ambiguous sentence.

There is a large body of work in the area of test-input generation. The rest of this section explores different techniques, where our focus lies on research that is concerned with generating test-data for compiler testing or research that is otherwise concerned with the generation of test-data that satisfies some precondition of a property.

9.1 Type-Driven Generation

Ruciman et al. [52] developed SmallCheck which, like QuickCheck, uses type-driven generators, but instead of generating test cases at random, it exhaustively tests the property for all input values up to some depth, progressively increasing the depth used. As such, a successful test-run gives the assurance that the specified property does not fail on any input bounded by this depth. This is in contrast to the approach taken in this thesis, where terms are not generated exhaustively and no such claim can be made.

Pałka et al. [47, 46] generate random well-typed lambda terms with the goal of testing an optimizing compiler. The terms are generated by interpreting the typing rules backwards: to generate a term that is in the consequence of a rule, it is necessary to generate terms that are in its premises. When multiple rules are applicable, the generator chooses a rule at random. A size limit is imposed to ensure that the generation terminates and the procedure backtracks whenever it goes astray. Fetscher et al. [19] generalize this technique to generate random well-typed terms based on the specification of a type system that is defined in Redex. Type judgments in Redex may refer to *metafunctions*, for example to look up the type of a name in the environment. To generate terms that satisfy arbitrary metafunctions, the metafunctions are compiled to judgment form. The clauses of a metafunction are ordered, requiring the addition of equational and disequational constraints as premises and a solver that is capable of solving these constraints. The approach described in this thesis is similar to the approach taken by Pałka et al. and Fetscher et al; for an in-depth discussion see Section 5.6.

Grieco et al. [22] developed QuickFuzz to fuzz software that manipulate complex file formats such as images. QuickFuzz automatically derives suitable Arbitrary instance declarations for a given type, which are used by QuickCheck's generators to generate random instances. Since Haskell's data types do not encode all invariants that should hold on values of a given data type, the generated instances are potentially invalid. This is different from our work, where no ill-formed programs are generated.

Midtgaard et al. [41] observed that with a type-directed approach many of the generated programs have output that depend on the evaluation order. For languages where the evaluation order is not specified, a generated program may produce different observable effects on different implementations. With such non-determinism it is no longer obvious how to check that the language implementation behaves as desired. To overcome this problem, Midtgaard et al. develop a type and effect system and that captures when the evaluation order is inconsequential for the observable behavior of the program. By generating terms from this type and effect system, only programs without observable evaluation-order dependence are generated.

9.2 Imperative Generation

Csmith [61] and its predecessor *randprog* [18] randomly generate C programs that conform to the C99 standard. Both tools avoid generating programs with undefined or unspecified behavior, because this would destroy the ability to automatically find bugs. To reach this goal, the generator structurally excludes certain programs from being generated and adds run-time checks to the generated programs to avoid operations that would potentially cause

undefined behavior. Compared to our work, Csmith is specifically tuned for the C programming language. Given that its implementation totals over 40,000 lines of C++ code, porting the technique to other programming languages would require a considerable engineering effort. Csmith also avoids generating programs with undefined or unspecified behavior, which is not the case for our generator. It would be interesting to explore a language-independent generator that avoids generating programs with undefined or unspecified behavior. Finally, it would be interesting to compare the performance of our language-independent generator on a definition of C to a language-specific generator for C.

ASTGen [13] is a framework for the automated generation of input programs that was created with the goal of testing refactoring engines. ASTGen allows developers to write imperative generators whose executions produce input programs. By combining primitive generators one can create a generator of more complex data. ASTGen exhaustively tests all inputs within a given bound (*bounded-exhaustive* generation). Compared to our work, ASTGen generates programs orders of magnitude faster, but it is not language-parametric and requires special care to avoid generating ill-formed programs.

9.3 Needed Narrowing

Lazy SmallCheck [52, 51] uses *needed narrowing* to repeatedly refine a partially defined value such that a precondition is satisfied. If the precondition succeeds (resp. fails) on the partially defined value, then it will also succeed (resp. fail) on all refinements of this value, which makes it possible to prune the search space on all equivalent values. If the precondition is undefined on the partially defined value, then the value is refined at exactly the place needed for evaluation of the condition to proceed further.

Fowler and Hutton [20] observe that in practice needed narrowing often leads to excessive backtracking resulting in poor efficiency. A naive needed narrowing strategy is to refine the data such that it satisfies the first condition before refining the data such that it also satisfies the second condition. This can lead to unnecessary backtracking, because as the data is being refined to satisfy the first condition it may already be obvious that the data will fail to satisfy the second condition. By evaluating all preconditions at once instead of one after another the needed narrowing strategy can backtrack as soon as the input data fails to satisfy any of the preconditions. For the purpose of generating well-typed terms this approach is similar to the approach taken in this thesis. In particular, our generator refines the partial program by solving constraints and backtracks when it realizes the partial program is ill-formed.

Lampropoulos et al. present *Luck*, a domain-specific language for writing property-based generators [35]. This language comes with a predicate semantics and a generator semantics, allowing a single artifact to be used as both a predicate and generator. The generator semantics in Luck can be annotated to control the distribution of the generated values and the amount of constraint solving that happens before a variable is instantiated.

9.4 Enumeration

Duregård et al. [17] present FEAT, a Haskell library that uses *functional enumerations* to efficiently compute a bijection from the natural numbers to a set of values by memoising the cardinalities of underlying sets, enabling the efficient computation of a value at an arbitrary index.

In the same spirit, SciFe [33] is a Scala framework for defining enumerators. SciFe provides a set of combinators to build larger enumerators out of smaller enumerators and introduces *higher-order enumerators* for defining enumerators that depend upon other enumerators, thereby providing the necessary expressiveness for defining enumerations of data structures that need to satisfy complex invariants. The ability to efficiently compute the value at an arbitrary index enables random generation.

Claessen et al. [12] build upon FEAT to generate constrained random data with a uniform distribution. Random indices are sampled from the set of natural numbers according to a uniform distribution and the precondition is evaluated on the value at this index. Every value, including the values that satisfy the predicate, has the same probability of being tested, resulting in a uniform distribution on the generated data. If the value does not satisfy the precondition it is discarded and all equivalent values are pruned from the search space, similar to Lazy SmallCheck.

These enumeration-based approaches differ from our approach in that they are unable to express complicated invariants such as type correctness of the enumerated terms. On the other hand, enumeration-based approaches provide more control over the distribution of the generated terms. Moreover, enumeration-based approaches allow structurally enumerating all terms bounded by a given size, which is not possible in our approach.

Chapter 10

Conclusions and Future Work

This chapter first gives an overview of the project's contributions. After this overview we reflect on the results of our evaluations and draw several conclusions. Finally, some ideas for future work are discussed.

10.1 Contributions

In this thesis we made the following contributions:

- The design and implementation of a language-parametric generator of well-formed sentences based on SDF (Section 3).
- An evaluation of the effectiveness of sentence generator at discovering ambiguities in an SDF3 grammar as well as discovering grammar differences (Section 4).
- The design and implementation of a language-parametric generator of well-formed terms based on NaBL2 (Section 5).
- An evaluation of the effectiveness of the term generator at discovering type soundness bugs and conformance bugs (Section 6).
- An analysis of why the generator fails to discover certain compiler bugs (Section 7).

10.2 Conclusions

First we presented a language-parametric algorithm for automatically generating sentences from an SDF3 grammar as well as an algorithm for automatically shrinking ambiguous sentences. We continued by presenting a language-parametric algorithm for automatically generator well-typed terms based on the specification of a programming language's syntax and static semantics (described in SDF3 and NaBL2). Finally, we evaluated the effectiveness of both generators at discovering different kinds of compiler bugs and analysed why certain compiler bugs are not caught within reasonable time.

Using our sentence generator we were able to find ambiguities in many SDF3 grammars with very little effort. Moreover, we used our sentence generator successfully to discover sentences that are accepted by an SDF3 grammar but should not be part of the language. Finally, the generated sentences successfully exposed several bugs in the meta-tools (Spoofax' pretty-printer, parser generator, and parser).

Using our term generator we were able to find a bug in all but two MiniJava compilers. We exposed a bug in a MiniJava compiler that was believed to be correct based on the manual test suite. We successfully used our term generator to discover type soundness bugs in simple programming language, but more research is necessary to discover type soundness bugs in more complex programming languages.

Though the idea of random program generation is not new, these results show that a lack of proper tool support is detrimental to the quality of software. We have shown that random compiler testing is a cost-effective method to find bugs in syntax definitions, verify meta-theoretic properties such as type soundness, and aid in the development of new languages. Given the little effort that is required to start using random testing, we believe that random testing should become a part of the standard process of language engineering.

10.3 Future Work

We have several ideas to make the sentence- and term generator more effective at discovering compiler bugs, as well as new applications of random testing. Most of these ideas are motivated by the discussion in Chapter 8.

- We created a sentence generator for SDF3, but there are many other grammar formalisms being used in practice. The vast number of grammar formalisms makes building separate sentence generators (and shrinkers) for every grammar formalism a large effort. Since most grammar formalisms are derived from BNF, it should be possible to ease the creation of a sentence generator for a new formalism. One way to achieve this is to transform various grammar formalisms to a more abstract core formalism and define the sentence generator in terms of this core formalism.

- We suspect that the cost of backtracking can be reduced significantly by generating independent parts of the program in isolation (as discussed in Chapter 8 under 'Exploit dependencies'). A first step would be to investigate the potential benefit of exploiting the dependencies in a language, for example by creating a language-specific generator for a simple language that has these dependencies encoded into the algorithm. If encoding the dependencies in the generator improves the performance, then the next step would be to generalize these insights and extract the dependencies from the NaBL2 specification.

- The generator is fairly naive when deciding which constraint to solve next and which alternative to continue with (in both cases, the generator choose uniformly at random). As a consequence, the generator frequently makes local choices that may never lead to a valid program, leading to excessive backtracking and eventually triggering

a timeout, causing the generator to start over. At the same time, the constraint generation rules provide a wealth of information about the language under test. Moreover, the current state of the constraint solver (such as which constraints need to be solved next) provide more information about which choices are more likely to yield a valid program. We suspect that the efficiency of the generator can be improved by making more informed choices.

- For the purpose of testing type soundness it makes sense to avoid programs with a lot of unreachable code. Since unreachable code is never evaluated, it is unlikely to trigger a type soundness bug. Our evaluation revealed several cases of unreachable code. For example, a MiniJava program starts executing at the main method, but if the main method does not invoke another method, then all code outside the main method is unreachable. Similarly, if a function in the lambda calculus is never applied then its body is never evaluated and thus unreachable. Finally, in many languages the evaluation of logical conjunction and disjunction are short-circuited/evaluated lazily. That is, the second operand is only evaluated if the first operand does not suffice to determine the value of the expression. We suspect that by avoiding these kinds of programs the generator can become more effective at discovering type soundness bugs.

- This thesis focused solely on generating well-formed programs, but certain applications could benefit from the generation of ill-formed programs. For example, when testing the conformance of a compiler to its specification, not only should well-typed programs be accepted by the type-checker, it should also reject ill-typed programs. How to efficiently generate ill-typed programs remains an open question.

- While evaluating student MiniJava compilers for conformance in Section 6.1, we noticed that a small number of compilers outputted the same erroneous result for some tests. Though there can be many explanations for this phenomenon, one could use this as an indicator for plagiarism. Specifically, by exposing student compilers to a large volume of test inputs and by automatically classifying compilers that make the same mistake, we suspect that random term generation can help detect plagiarism.

10.4 Source Code

The source code of the sentence generator as well as the code used for evaluating the sentence generator is available at `https://github.com/metaborg/spg`. The languages that were used to detect ambiguities can be found at `https://github.com/spg-subjects`.

The source code of the term generator is available at `https://github.com/MartijnDwars/spg`. This repository also contains references to the languages that were used for evaluating the term generator and the infrastructure that was used for the evaluation.

Bibliography

[1] SDF Disambiguation. `http://www.meta-environment.org/doc/books/syntax/sdf-disambiguation/sdf-disambiguation.html`. Accessed: 2017-05-02.

[2] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[3] Hendrik van Antwerpen, Pierre Néron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A Constraint Language for Static Semantic Analysis based on Scope Graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 49–60. ACM, 2016.

[4] Andrew W. Appel. *Modern Compiler Implementation [in ML] [in C] [in Java]*. Cambridge University Press, 1998.

[5] Bas Basten and Jurgen J. Vinju. Faster Ambiguity Detection by Grammar Filtering. In Claus Brabrand and Pierre-Etienne Moreau, editors, *Proceedings of the of the Tenth Workshop on Language Descriptions, Tools and Applications, LDTA 2010, Paphos, Cyprus, March 28-29, 2010 - satellite event of ETAPS*, page 5. ACM, 2010.

[6] H. J. S. Basten and Jurgen J. Vinju. Parse Forest Diagnostics with Dr. Ambiguity. In *Proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011)*, volume 6940 of *LNCS*. Springer, July 2011.

[7] Franco Bazzichi and Ippolito Spadafora. An Automatic Generator for Compiler Testing. *IEEE Transactions on Software Engineering*, (4):343–353, 1982.

[8] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley Professional, 2003.

[9] Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing Ambiguity of Context-Free Grammars. 75(3), March 2010. Earlier version in Proc. 12th International Conference on Implementation and Application of Automata, CIAA '07, Springer-Verlag LNCS vol. 4783.

[10] Augusto Celentano, Stefano Crespi-Reghizzi, Pierluigi Della Vigna, Carlo Ghezzi, G. Granata, and F. Savoretti. Compiler Testing using a Sentence Generator. *Software: Practice and Experience*, 10(11):897–918, 1980.

[11] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: A Software Testing Service. *Operating Systems Review*, 43(4):5–10, 2009.

[12] Koen Claessen, Jonas Duregård, and Michał H. Pałka. Generating Constrained Random Data with Uniform Distribution. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2014.

[13] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated Testing of Refactoring Engines. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 185–194. ACM, 2007.

[14] Luís Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser. Principled Syntactic Code Completion using Placeholders. In Tijs van der Storm, Emilie Balland, and Dániel Varró, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pages 163–175. ACM, 2016.

[15] Saumya K. Debray, William S. Evans, Robert Muth, and Bjorn De Sutter. Compiler Techniques for Code Compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 2000.

[16] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Fuzzing The Rust Typechecker Using CLP. In Myra B. Cohen, Lars Grunske, and Michael Whalen, editors, *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 482–493. IEEE, 2015.

[17] Jonas Duregård, Patrik Jansson, and Meng Wang. FEAT: Functional Enumeration of Algebraic Types. In Janis Voigtländer, editor, *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*, pages 61–72. ACM, 2012.

[18] Eric Eide and John Regehr. Volatiles Are Miscompiled, and What to Do about It. In Luca de Alfaro and Jens Palsberg, editors, *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, Atlanta, GA, USA, October 19-24, 2008*, pages 255–264. ACM, 2008.

[19] Burke Fetscher, Koen Claessen, Michał H. Pałka, John Hughes, and Robert Bruce Findler. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In Jan Vitek, editor, *Programming Languages*

and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, volume 9032 of *Lecture Notes in Computer Science*, pages 383–405. Springer, 2015.

[20] Jonathan Fowler and Graham Hutton. Failing Faster: Overlapping Patterns for Property-Based Testing. In Yuliya Lierler and Walid Taha, editors, *Practical Aspects of Declarative Languages - 19th International Symposium, PADL 2017, Paris, France, January 16-17, 2017, Proceedings*, volume 10137 of *Lecture Notes in Computer Science*, pages 103–119. Springer, 2017.

[21] Michaela Greiler, Arie van Deursen, and Margaret-Anne D. Storey. Automated Detection of Test Fixture Strategies and Smells. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, Luxembourg, March 18-22, 2013*, pages 322–331. IEEE, 2013.

[22] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. QuickFuzz: An Automatic Random Fuzzer for Common File Formats. In Geoffrey Mainland, editor, *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, pages 13–20. ACM, 2016.

[23] Kenneth V. Hanford. Automatic Generation of Test Cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[24] Jan Heering, P. R. H. Hendriks, Paul Klint, and Jan Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[25] John Hughes. Software Testing with QuickCheck. In Zoltán Horváth, Rinus Plasmeijer, and Viktória Zsók, editors, *Central European Functional Programming School - Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures*, volume 6299 of *Lecture Notes in Computer Science*, pages 183–223. Springer, 2009.

[26] Lennart C. L. Kats, Rob Vermaas, and Eelco Visser. Integrated Language Definition Testing: Enabling Test-Driven Language Development. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 139–154. ACM, 2011.

[27] Lennart C. L. Kats, Rob Vermaas, and Eelco Visser. Testing Domain-Specific Languages. In Cristina Videira Lopes and Kathleen Fisher, editors, *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 25–26. ACM, 2011.

[28] Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In William R. Cook, SiobhÃąn Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM.

[29] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run Your Research: On The Effectiveness of Lightweight Mechanization. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 285–296. ACM, 2012.

[30] Casey Klein and Robby Findler. Randomized Testing in PLT Redex. In *Workshop on Scheme and Functional Programming (SFP)*, 2009.

[31] Casey Klein, Matthew Flatt, and Robert Bruce Findler. The Racket virtual machine and randomized testing. *Higher-Order and Symbolic Computation*, 25(2-4):209–253, 2012.

[32] Claessen Koen and Hughes John. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proc. of International Conference on Functional Programming (ICFP), ACM SIGPLAN*, pages 268–279, 2000.

[33] Ivan Kuraj and Viktor Kuncak. SciFe: Scala Framework for Efficient Enumeration of Data Structures with Invariants. In Philipp Haller and Heather Miller, editors, *Proceedings of the Fifth Annual Scala Workshop, SCALA@ECOOP 2014, Uppsala, Sweden, July 28-29, 2014*, pages 45–49. ACM, 2014.

[34] Ralf Lämmel. Grammar Testing. In Heinrich HuÃ§mann, editor, *Fundamental Approaches to Software Engineering, FASE 2001*, volume 2029 of *Lecture Notes in Computer Science*, pages 201–216. Springer, 2001.

[35] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. Beginner's Luck: A Language for Property-Based Generators. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 114–129. ACM, 2017.

[36] Christian Lindig. Random Testing of C Calling Conventions. In Clinton Jeffery, Jong-Deok Choi, and Raimondas Lencevicius, editors, *Proceedings of the Sixth International Workshop on Automated Debugging, AADEBUG 2005, Monterey, California, USA, September 19-21, 2005*, pages 3–12. ACM, 2005.

[37] Brian A Malloy and James F Power. An Interpretation of Purdom's Algorithm for Automatic Generation of Test Cases. 2001.

[38] Brian A Malloy and James F Power. A Top-Down Presentation of Purdom's Sentence-Generation Algorithm. *Maynooth, Co. Kildare, Ireland*, 2005.

[39] William M McKeeman. Differential Testing for Software. *Digital Technical Journal*, 10(1):100–107, 1998.

[40] Bruce McKenzie. Generating strings at random from a context free grammar. 1997.

[41] Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. Effect-Driven QuickChecking of Compilers. *PACMPL*, 1(ICFP), 2017.

[42] Jan Midtgaard and Anders Moller. QuickChecking Static Analysis Properties. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–10. IEEE, 2015.

[43] V. Murali and R. K. Shyamasundar. A Sentence Generator for a Compiler for PT, a Pascal Subset. *Software: Practice and Experience*, 13(9):857–869, 1983.

[44] Glenford J. Myers. *The Art of Software Testing (2. ed.)*. Wiley, 2004.

[45] Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A Theory of Name Resolution. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015.

[46] Michał H Pałka. *Random Structured Test Data Generation for Black-Box Testing*. PhD thesis, 2014.

[47] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 91–97, New York, NY, USA, 2011. ACM.

[48] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, 2002.

[49] Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[50] Paul Purdom. A Sentence Generator for Testing Parsers. *BIT Numerical Mathematics*, 12:366–375, 1972. 10.1007/BF01932308.

[51] 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 - 23rd International Symposium, IFL 2011, Lawrence, KS, USA, October 3-5, 2011, Revised Selected Papers*, volume 7257 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 2011.

[52] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In Andy Gill, editor, *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, pages 37–48. ACM, 2008.

[53] Pawel Urzyczyn. Inhabitation in Typed Lambda-Calculi (A Syntactic Approach). In Philippe de Groote, editor, *Typed Lambda Calculi and Applications, Third International Conference on Typed Lambda Calculi and Applications, TLCA 97, Nancy, France, April 2-4, 1997, Proceedings*, volume 1210 of *Lecture Notes in Computer Science*, pages 373–389. Springer, 1997.

[54] Mark G. J. van den Brand and Eelco Visser. Generation of Formatters for Context-Free Languages. *ACM Transactions on Software Engineering Methodology*, 5(1):1–41, 1996.

[55] Vlad A. Vergu, Pierre Néron, and Eelco Visser. DynSem: A DSL for Dynamic Semantics Specification. In Maribel Fernández, editor, *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*, volume 36 of *LIPIcs*, pages 365–378. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[56] Jurgen J. Vinju. SDF Disambiguation Medkit for Programming Languages. Technical Report SEN-1107, Centrum Wiskunde & Informatica, april 2011. Appeared in 2006 online at http://www.meta-environment.org, and was published as CWI technical report in 2011.

[57] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[58] Eelco Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In Aart Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–362. Springer, 2001.

[59] Tobi Vollebregt, Lennart C. L. Kats, and Eelco Visser. Declarative Specification of Template-Based Textual Editors. In Anthony Sloane and Suzana Andova, editors, *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*, pages 1–7. ACM, 2012.

[60] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 115(1):38–94, November 1994.

[61] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011.

Appendix A

Grammar Differences

This appendix lists the differences between the SDF3 grammar and ANTLRv4 grammar for Pascal and Java 8 that we identified while evaluating the effectiveness of the sentence generator at discovering differences between grammars in Section 4.3. We relate each difference to the language specification to identify whether it is a bug in the SDF3 grammar (indicated by **Bug in SDF3 grammar**) or a bug in the ANTLRv4 grammar (indicated by **Bug in ANTLRv4 grammar**).

A.1 Pascal

We discovered 19 differences between the SDF3 grammar [1]) and ANTLRv4 grammar [2] for Pascal. By comparing these differences to the ISO 7185:1990 language specification, we were able to identify 15 out of 19 differences as bugs in the SDF3 grammar. The remaining 4 differences are bugs in the ANTLRv4 grammar.

1. **Bug in SDF3 grammar** The SDF3 grammar allows a program heading with an empty list of parameters. According to the standard, the list should either be omitted, or the list should contain one or more parameters.

2. **Bug in SDF3 grammar** The SDF3 grammar allows label declarations that are integer numbers which can contain a sign prefix. According to the standard, a label may not contain a sign prefix.

3. **Bug in SDF3 grammar** Pascal has a set-type that is parameterized with a base-type. A base-type is an ordinal-type, which is a subset of all types. The SDF3 grammar allows a set-type to be parameterized by any type.

4. **Bug in SDF3 grammar** The SDF3 grammar allows the scale factor of a real number to contain two plus symbols, e.g. `484e++610`. This is not allowed by the standard.

[1] `https://github.com/spg-subjects/metaborg-pascal` (commit c54654a)
[2] `https://github.com/antlr/grammars-v4/` (commit 5c680cd)

5. **Bug in SDF3 grammar** The SDF3 grammar allows a procedure call with an empty list of actual arguments. According to the standard, the list of parameters should either be omitted or contain at least one actual argument.

6. **Bug in SDF3 grammar** In Pascal, a record type contains a field list that can be omitted or contain at least one fixed part. The SDF3 grammar allows an empty field list.

7. **Bug in SDF3 grammar** The SDF3 grammar requires a semicolon after every record section in the field list. The ANTLRv4 grammar uses a semicolon as separator. According to the standard, the last semicolon is optional.

8. **Bug in SDF3 grammar** The standard a comma separated list of index expressions when accessing an array. SDF3 allows zero index expressions.

9. **Bug in SDF3 grammar** According to the standard, the `case-list-elements` in a `case-statement` are separated by a semicolon and the last element may end with a semicolon. The ANTLRv4 grammar *does not allow* an optional trailing semicolon. The SDF3 grammar *requires* a trailing semicolon.

10. **Bug in SDF3 grammar** According to the standard, nested statements may not be labeled. The SDF3 grammar allows nested labeled statements.

11. **Bug in SDF3 grammar** According to the standard, the literal `'not'` can only precede a `factor` and a `factor` cannot derive a string that starts with a minus sign. The SDF3 grammar allows the derivation of the expression `not - true`.

12. **Bug in SDF3 grammar** In SDF3, an expression can be prefixed by a sign, the result of which is itself an expression. Specifically, SDF3 considers `+-false` to be a valid expression. According to the standard, an expression that starts with a sign can only be followed by a term, and a term cannot start with a sign.

13. **Bug in SDF3 grammar** A record type can have variants. In the SDF3 grammar, the variants need to end with a semicolon. In the ANTLRv4 grammar, the variants must not end with a semicolon. In the ISO7185 standard, the variants may optionally end with a semicolon.

14. **Bug in SDF3 grammar** The SDF3 grammar supported an optional statement at the end of the block, which is non-standard.

15. **Bug in SDF3 grammar** The SDF3 grammar requires a semicolon after every RecordSection. The ANTLRv4 grammar requires no semicolon after the last RecordSection. The specification has an optional semicolon after the last RecordSection.

16. **Bug in ANTLRv4 grammar** The SDF3 grammar supports the conformant-array-parameter. The ANTLRv4 grammar does not support this kind of parameter type.

17. **Bug in ANTLRv4 grammar** In the SDF3 grammar, a formal parameter can be a procedure heading (resp. function heading) which can in turn be a formal parameter again. As a result, procedure headings can be nested arbitrarily deep. In the ANTLRv4 grammar, this nesting can only go one level deep.

18. **Bug in ANTLRv4 grammar** The SDF3 grammar supports procedure declarations consisting of a procedure heading followed by a directive (but no body). The ANTLRv4 grammar does not support this construct.

19. **Bug in ANTLRv4 grammar** The SDF3 grammar allows forward function declarations (function declaration without parameters and return type). The ANTLRv4 grammar does not. According to the specification, forward function declarations should be allowed.

A.2 Java 8

We discovered 16 differences between the SDF3 grammar [3] and the ANTLRv4 grammar [4] for Java. By comparing these differences to the JLS 1.8 we can attribute 13 out of these 16 differences to bugs in the SDF3 grammar. The remaining differences are due to bugs in the ANTLRv4 grammar.

1. **Bug in SDF3 grammar** In the SDF3 grammar a try-with-resources statement can have a resource clause that consists solely of a semicolon. According to the language specification, the resource specification should be a comma-separated list of one or more resources.

2. **Bug in SDF3 grammar** In the SDF3 grammar a try-with-resources statement can have an empty resource clause. According to the specification, the resource clause should contain at least one resource.

3. **Bug in SDF3 grammar** In the SDF3 grammar an element value can be an expression, but in the specification an element value can only be a *conditional* expression. A conditional expression is a subset of all expressions. Specifically, assignment operators such as *=, /=, and += are not allowed in a conditional expression.

4. **Bug in SDF3 grammar** In the SDF3 grammar a `Statement` can be an `Expression` followed by a semicolon. According to the specification, only `ExpressionStatements` followed by a semicolon are valid statements. In particular, the `this` expression is a valid `Expression` but not a valid `ExpressionStatement`.

5. **Bug in SDF3 grammar** In the SDF3 grammar a `ConstantDeclaration` may not end with a semicolon. According to the specification, a `ConstantDeclaration` must end with a semicolon.

[3] `https://github.com/spg-subjects/java-front` (commit 78d53cb)
[4] `https://github.com/antlr/grammars-v4/` (commit 5c680cd)

6. **Bug in SDF3 grammar** In the SDF3 grammar a `ForInit` can be a list of `Expressions`, but according to the specification, this should be a list of `StatementExpressions`. The latter does not contain, for example, pre-increment expressions.

7. **Bug in SDF3 grammar** In the SDF3 grammar a method reference expression can start with an arbitrary expression. According to the specification, only expression names and primary expressions (which include only the simplest kinds of expressions) are allowed.

8. **Bug in SDF3 grammar** In the SDF3 grammar a lambda expression can be postfix-incremented. According to the specification, a `PostIncrementExpression` can be a primary expression (among others), but not a lambda expression.

9. **Bug in SDF3 grammar** In the SDF3 grammar the right hand side of a conjunction can be an arbitrary expression. According to the specification, the right hand side of a conjunction must be a symbol of sort `InclusiveOrExpression`. In particular, a lambda expression is not allowed on the right hand side of a conjunction.

10. **Bug in SDF3 grammar** In the SDF3 grammar an enum declaration can start with a comma. According to the specification, this is not allowed.

11. **Bug in SDF3 grammar** In the SDF3 grammar a post-increment expression could not be nested within a post-decrement expression (or vice versa). This was caused by a context-free priority group that specified post-increment and post-decrement expressions as right-associative, even though there was no ambiguity.

12. **Bug in SDF3 grammar** In the SDF3 grammar a receiver parameter may refer to a top-level class using syntax <ID>.class. According to the standard, the dot may be surrounded by whitespace, whereas the SDF3 grammar did not allow whitespace around the dot.

13. **Bug in SDF3 grammar** In the SDF3 grammar a formal parameter list can start with a receiver parameter followed by a comma, followed by zero or more formal parameters, followed by a comma, followed by the last formal parameter. If there are zero formal parameters, the SDF3 grammar expects two adjacent commas, which it not consistent with the standard.

14. **Bug in ANTLRv4 grammar** In the SDF3 grammar a Unicode escape sequence must start with '\u' followed by four hexadecimal digits, e.g. '\u0123'. The ANTLRv4 grammar only allows a single 'u' character instead of one or more 'u' characters. The SDF3 grammar is consistent with the specification, hence this is a bug in the ANTLRv4 grammar.

15. **Bug in ANTLRv4 grammar** In the ANTLRv4 grammar a method's formal parameter list may not contain a single receiver parameter. The SDF3 grammar allows a single receiver parameter, which is consistent with the specification. Hence this is a bug in the ANTLRv4 grammar.

16. **Bug in ANTLRv4 grammar** The ANTLRv4 grammar allows whitespace between the less-than and greater-than sign of a shift expression. For example, `x < < 1` is a valid shift expression in the ANTLRv4 grammar. The SDF3 grammar does not allow whitespace between these signs, which is consistent with the specification.

Appendix B

Mutants

For evaluating the effectiveness of the randomly generated terms at discovering type soundness bugs we introduced various bugs into existing languages. By carefully introducing these bugs we create a mutation of the original language that is correct except for the bug we introduced; we refer to these languages as *mutants*. This appendix describes the six mutants that we created in L1, L2, and L3 and then describes the thirteen mutants in Tiger.

B.1 L1-2-3 Mutants

We created six mutants of L1, L2, and L3 that each make the type system unsound with respect to the operational semantics. Mutants 1, 2, 3, 5, and 6 involve a modification to the type system (encoded in NaBL2). Mutant 4 involves a modification to the operational semantics (encoded in DynSem).

1. A function with argument type τ_1 and body type τ_2 has type $\tau_2 \rightarrow \tau_1$ instead of $\tau_1 \rightarrow \tau_2$.
2. A function does not scope its body, but instead the body of a function shares the surrounding scope.
3. An integer literal has an arbitrary type.
4. A function does not close over its environment (i.e. act as a closure).
5. The type of an application of a function $\tau_1 \rightarrow \tau_2$ to an argument of type τ_1 is τ_1 (i.e. types swapped).
6. The first operand of an application has type τ_1 and the second operand has type $\tau_1 \rightarrow \tau_2$ (i.e. types swapped).

B.2 Tiger Mutants

We created thirteen Tiger mutants by injecting the following errors:

1. No (sub)type check on the arguments of a call.
2. Boolean operators (`Or/2`, `And/2`) operate on arbitrary (but equal) types.
3. Type of the parameter list `[a1, ..., an]` of a function is the type of `[a1]`.
4. No (sub)type check for variable declarations.
5. Branches of `If/3` are not required to have the same type.
6. `Subscript/2` is always of type `INT()`, independent of the underlying array.
7. `Array/3` is always of type `ARRAY(INT(), _)`, independent of the actual type.
8. DynSem misses implementation for `Eq` on `UnitV()`
9. Implement `Break` as value instead of as exception.
10. `Assign/2` has the type of the assigned expression instead of unit.
11. `Let/2` does not scope its body.
12. `VarDecl/2` has *letrec* semantics.
13. No `nil`-check when accessing a field of a record.

Appendix C

Generated Programs

To get an impression of the programs that are being generated we list a small sample of 10 programs generated for L1, one of the languages that were used in the evaluation. The generator was invoked with configuration parameters $maxSize = 100$ and $steps = 500$.

```
fun(n177: Int) {
    fun(n176: Int) {
        5
    }
}
```

```
fun(n271: (Int -> ((Int -> Int ->
↪  Int) -> Int -> Int) -> (Int
↪  -> Int) -> (((Int -> Int) ->
↪  (Int -> Int) -> Int) -> Int)
↪  -> Int) -> Int) {
    -9
}
```

```
-4
```

```
fun(n147: (Int -> Int) -> Int) {
    n147
}
```

```
fun(n151: ((((Int -> Int -> ((Int
↪  -> Int) -> (((Int -> Int) ->
↪  Int -> Int) -> Int) -> Int)
↪  -> Int) -> (((Int -> Int) ->
↪  Int) -> Int -> (((Int -> Int)
↪  -> Int) -> (((Int -> (Int ->
↪  Int) -> Int -> Int) -> Int)
↪  -> Int) -> Int) -> (Int ->
↪  Int) -> Int) -> Int -> Int)
↪  -> Int -> Int -> Int) -> Int
↪  -> Int) -> Int) {
    n151
}
```

```
fun(n235: Int) {
    n235
}(-32954) + 066
```

```
fun(n214: Int) {
    fun(n473: Int) {
        fun(n472: Int -> Int) {
            -0
        }(fun(n471: Int) {
            -5
        })
    }(fun(n470: Int) {
```

```
        n470
    }(n214) + n214) + n214
}
```

```
531
```

```
fun(n433: Int -> ((Int -> Int ->
↪   Int) -> (((Int -> (Int ->
↪   Int) -> ((((Int -> Int) ->
↪   (Int -> Int) -> Int) -> Int)
↪   -> Int) -> Int -> (Int ->
↪   Int) -> (Int -> Int) -> (Int
↪   -> Int) -> Int -> (Int ->
↪   Int) -> Int) -> Int -> Int)
↪   -> Int) -> Int) -> ((Int ->
↪   Int) -> Int) -> Int) {
    fun(n432: Int) {
```

```
        -8
    }
}
```

```
fun(n180: Int -> Int) {
    fun(n311: Int -> Int) {
        fun(n310: Int -> Int) {
            -962
        }
    }(n180)(n180)
}
```