Thesis for the Degree of Doctor of Philosophy

Lightweight Verification of Functional Programs

Nicholas Smallbone

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2013

Lightweight Verification of Functional Programs
Nicholas Smallbone

ISBN 978-91-7385-841-0

© 2013 Nicholas Smallbone

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 3522
ISSN 0346-718X
Technical Report 95D
Department of Computer Science and Engineering
Research group: Functional Programming

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers
Göteborg, Sweden 2013

Abstract

We have built several tools to help with testing and verifying functional programs. All three tools are based on QuickCheck properties. Our goal is to allow programmers to do more with QuickCheck properties than just test them.

The first tool is QuickSpec, which finds equational specifications, and can be used to help with writing a specification or for program understanding. On top of QuickSpec, we have built HipSpec, which proves properties about Haskell programs, and uses QuickSpec to prove the necessary lemmas. We also describe PULSE and `eqc_par_statem`, which together can be used to find race conditions in Erlang programs.

We believe that testable properties are a good basis for reasoning and verification, and that they give many of the benefits of formal verification without the cost of proof. The chief reason is that they are formal specifications for which the programmer can always get a counterexample when they are false. Furthermore, using testable properties allows us to write better tools. None of our tools would be possible if our properties were not testable.

We also present work on encoding types in first-order logic, an essential component when using first-order provers to reason about programs. Our encodings are simple but extremely efficient, as evidenced by benchmarks. We develop the theory behind sound type encodings, and have written tools that implement our ideas.

Contents

Introduction 1

I Discovering and proving properties

Paper 1 QuickSpec: Formal Specifications for Free! 24

1 Introduction 24
2 How QuickSpec Works 33
3 Case Studies 50
4 Related Work 64
5 Conclusions and Future Work 66

Paper 2 HipSpec: Automating Inductive Proofs using Theory Exploration 69

1 Introduction 69
2 Implementation 73
3 Examples 76
4 Evaluation 79
5 Related Work 84
6 Conclusion and Further Work 86

II Encoding types in first-order logic

Paper 3 Sort It Out with Monotonicity: Translating between Many-Sorted and Unsorted First-Order Logic 88

1 Introduction 89
2 Monotonicity Calculus for First-Order Logic 93
3 Monotonox: Sorted to Unsorted Logic and Back Again 102
4 Translating Sorted to Unsorted Logic 104
5 Translating Unsorted to Sorted Logic 105

6 Results . . . 106
7 Conclusions and Future Work . . . 106

Paper 4 Encoding Monomorphic and Polymorphic Types 109
1 Introduction . . . 109
2 Background: Logics . . . 112
3 Traditional Type Encodings . . . 117
4 Monotonicity-Based Type Encodings—The Monomorphic Case . . . 123
5 Monotonicity-Based Encoding of Polymorphism . . . 135
6 Alternative, Cover-Based Encoding of Polymorphism . . . 143
7 Implementation . . . 148
8 Evaluation . . . 152
9 Related Work . . . 157
10 Conclusion . . . 158

III Finding race conditions

Paper 5 Finding Race Conditions in Erlang with QuickCheck and PULSE 161
1 Introduction . . . 161
2 Our Case Study: the Process Registry . . . 163
3 An Overview of Quviq QuickCheck . . . 165
4 Parallel Testing with QuickCheck . . . 169
5 PULSE: A User-level Scheduler . . . 174
6 Visualizing Traces . . . 181
7 Discussion and Related Work . . . 188
8 Conclusions . . . 193

Paper 6 Accelerating Race Condition Detection through Procrastination 196
1 Introduction . . . 197
2 Background . . . 199
3 Procrastination . . . 202
4 The ProcReg Example . . . 205
5 Procrastinating ProcReg . . . 207
6 Shrinking Counterexamples . . . 212
7 Related Work . . . 217
8 Conclusions . . . 218

Bibliography 220

Acknowledgements

I would like to thank my supervisor, Koen Claessen. Koen, you have been brilliant! Thanks for many engrossing discussions and crazy ideas and all your encouragement.

Thanks to all my collaborators and coauthors for all the exciting research! Also for putting up with my phenomenal procrastination at times… It has been fun!

To my colleagues at the department, who make this a wonderful place to work. Thanks especially to my friends here, for cheering me up when I've been stressed, and all the fun we've had together. I am planning to stay here for a while yet!

Thanks to my family for everything. To my grandma and grandad: I miss you. Lastly, my thanks go to my big brother, for turning me into a maths geek in the first place.

Introduction

Programmers spend half their time finding and fixing bugs [Myers and Sandler, 2004]. Even so, almost all software is buggy, much of it embarrassingly so. A typical, released piece of software has between 1 and 25 bugs per thousand lines of code [McConnell, 2004]. We are bad at writing correct software.

Why is it so hard to avoid bugs?

- *Software is complex.* Many bugs only appear under very specific conditions, and can go unnoticed for years, or worse, appear sporadically and mysteriously. The company that made the Therac-25, a radiotherapy machine which sometimes gave out lethal doses of radiation, flatly denied that their machine had a fault because they couldn't reproduce it.

- *Programs' requirements often change.* A correct program may become wrong when we use it in a different context. Or we may introduce a bug while updating the software. The Ariane 5 rocket crashed because it reused code from the Ariane 4, but the assumptions behind that code were no longer true.

- *Most programs are poorly specified.* It is hard to say if a program is correct if you can't even say what it should do! If different parts of the program disagree about what they should be doing, the parts might be correct in isolation but their combination will be wrong. The Mars Climate Orbiter crashed onto Mars because one module was using imperial units and the other metric.

One approach is to give up: let bugs happen, and fix them as users find them. But the earlier we find a bug, the easier it is to fix: a design flaw can cost 100 times more to fix once it reaches production than if it had been found during the design stage [McConnell, 2004]. A 2002 study [Tassey, 2002] found that bugs cost the US economy $60 billion a year. Bugs are expensive, and we must find them.

Testing The dominant method of finding bugs is *testing*. Typically, the programmer or tester writes down a lot of different test cases that express everything the software is supposed to do, and collects them into a test suite.

This is not a nice way of finding bugs. For one thing, inventing test cases is a dreadful bore, and many programmers will do their best to avoid it. This means software that's tested as little as possible, as late as possible—a surefire recipe for bugs! One way to solve this is test-driven development, which asks us to write the test suite before the program, and to never write a single line of code unless it makes a failing test pass. This makes sure that the test suite exists and covers all the functionality of the program.

A worse problem is that good test suites are very hard to write. It is too easy to miss an unusual case—and unusual cases are where the bugs are! Design flaws, especially, are likely to manifest themselves as unusual cases that do not work and cannot be fixed. Common wisdom suggests using a coverage tool to check that the test suite fully exercises the program. But code coverage can give a false sense of security [Marick, 1999], and a feeble test suite can have 100% coverage. A programmer who sees an uncovered line of code may write a test case just to cover that line—disastrous for test quality! Mutation testing [Offutt, 1994] measures the quality of the test suite by randomly mutating the program, in the hope of breaking it, and checking that the test suite finds a bug—but there is always the chance that the mutated program behaves the same as the original one, which makes it hard to interpret the results. Measuring the quality of a test suite is difficult.

Finally, many bugs are hard to provoke, and only appear in very particular circumstances. For example, to provoke a race condition you may have to insert delays into your code, while if an action breaks the internal state of the program, it may take several more steps before the program goes wrong [Zeller, 2005, chapter 1]. Finding a test case that provokes such a bug can be difficult, even when you know it exists, and writing a good test suite under such circumstances is very difficult.

Our position is that traditional test suites are underpowered for ensuring program correctness. They place a large manual burden on the tester, and do not give much confidence in the program afterwards. As Dijkstra [1972] said, "Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."

Proof Speaking of Dijkstra, we could always prove our programs correct. There are several approaches. Dijkstra advocated programs that are correct by construction: you should not take an existing program and try to prove it correct, but use mathematical reasoning to arrive at a program and its proof at the same time. This means that you will never write a program that you cannot prove correct.

Dijkstra only wrote pencil and paper proofs. These are excellent for program understanding, but run the risk that you make a mistake in your proof. This is more likely in programming than in mathematics, because proofs of programs tend to be simple but fiddly. There is also the danger of proving something about an idealised program rather than the one you have really written: for a long time, the Java standard library included a binary search algorithm proved correct by Bentley [1986], but this proof assumed that integers were unbounded, leading to a bug. If the program's requirements change, you have to work out what parts of the proof are invalidated, which is error-prone. Computer-checked proofs are much more convincing.

A more recent approach, perhaps the closest in spirit to Dijkstra's, is to write your program in a dependently-typed programming language, and encode the specification in the types. When it typechecks it must meet the specification. The most famous example of this is CompCert [Leroy, 2009], a verified C compiler written in Coq [The Coq development team, 2004]. CompCert is intended for critical software where the programmers would otherwise have to read the generated assembly code and check that it's correct, so a formal proof of correctness is a big advantage here.

Another success story for formal proof is the L4.verified project [Klein et al., 2010], which has proved a real operating system microkernel correct in Isabelle [Nipkow et al., 2002]. The proof is long—200,000 lines of Isabelle code to verify 7,500 lines of C code—but given the low-level, highly concurrent and dangerous nature of the code, this is an impressive achievement. Despite the difficulty of the proof, it was cheaper than certifying the software in the normal way.

Both CompCert and L4.verified rely on manual proofs performed using a proof assistant. Most programmers will not be willing to write long proofs by hand, and for most programs a semi-automatic approach is easier. Thanks to the success of SMT solvers, this is practical nowadays. Typically, the programmer specifies preconditions, postconditions and invariants, and the system generates verification conditions and sends them to an SMT solver such as Z3 [de Moura and Bjørner, 2008]. This works well since the verification conditions tend to be simple, and

proof is practical for programmers who are willing to annotate their programs. Systems in this class include Why3 [Bobot et al., 2011] and Boogie [Barnett et al., 2005], which can be used to verify programs written in a variety of languages.

Proof gives a large amount of confidence in the correctness of the program and does not miss edge cases or design flaws. The only trouble with proof is that *very few people do it!*

Why is this? Many programmers find writing test cases as enjoyable as a trip to the dentist; you would think they would appreciate a more powerful method. An obvious reason is that formal proof is expensive: for most projects, the increased reliability does not justify the cost. This is true, but even so, you would expect programmers to routinely use proof tools where they're most useful, for designing tricky data structures and the like.

We think there are two important reasons:

1. Most programs are not specified formally, so what should we prove?

2. Whatever we try to prove will probably not be true. The program will surely have a bug; there is a good chance the specification will be wrong too [Claessen and Hughes, 2000, §6.6]!

These two combine to create a big barrier to formal proof. First, the programmer has to spend time thinking up a property they expect to hold. The property will normally turn out to be false, and most proof tools do not detect false properties. So the poor programmer will give the property to the proof tool, which will respond with a curt "don't know" or by running forever. This is dreadful!

We argue that any verification tool *must* detect false properties and provide useful feedback, ideally a counterexample to the property. Verification tools must first of all find bugs! Proof is only useful for correct programs, and most programs are not correct. If the only reason to specify a program is to prove it correct, then for most programs there is no tangible benefit to formal specification.

Lightweight verification How can we provide useful feedback when a property is false?

A good example is a *typechecker*. A typechecker is a verification tool, even though many people don't think of it as one. Its job is to prove that the program cannot encounter a type error at runtime. If

typechecking fails, we don't get a counterexample, and the program might even be correct; nevertheless, we can point at a particular bit of the program where something suspicious happens.

The ESC/Modula-3 system [Leino and Nelson, 1998] pioneered the same approach for more powerful properties: don't provide an explicit counterexample, but point the user at suspicious behaviour. This works nicely for properties like array bounds checks, where a failed proof normally indicates a false property: you rarely need a complex argument to show why an array access can't be out of bounds. If you do, then the programmer should think extra hard about that access anyway, so a false positive is not very harmful.

Some proof tools do try to provide counterexamples for false properties. Systems based on SMT solvers will get a countermodel when the property is false; Boogie has a symbolic debugger [Le Goues et al., 2011] which translates these countermodels into source-language counterexamples. Isabelle has a counterexample finder, Nitpick [Blanchette and Nipkow, 2010], but since it works in higher-order logic it does not always succeed.

In general, it is easier to provide feedback the less expressive our property language is, and the less ambitious our properties are. Expressive property languages are bad for automatic tools!

Property-based testing We unfairly wrote off testing earlier: we can certainly do better than collecting hundreds of individual test cases.

QuickCheck [Claessen and Hughes, 2000] has pioneered *property-based testing*. In property-based testing, the programmer writes *properties* instead of test cases. In its most basic form a property is a boolean expression that should be true, just like a test case, but unlike a test case it may contain variables. QuickCheck will generate random values for those variables and check that the property always evaluates to true. For example, to test that a sorting function always returns a sorted list, we would write:

```
prop_sort :: [Int] -> Bool
prop_sort xs = sorted (sort xs)
```

QuickCheck will then check that `prop_sort xs` holds for randomly-chosen values of `xs`.

Property-based testing solves the problems of test cases that we mentioned earlier. Typically, a few properties can replace a large test suite, so testing is no longer a bore. Because we test random inputs,

we will not overlook edge cases, and we never have to dream up the right test case to provoke a tricky bug. We have the full power of the programming language at our disposal, so we can easily write *test oracles* that decide whether a test has passed—vital for testing non-deterministic and concurrent software. QuickCheck has proved wildly successful at finding bugs in a large number of well-tested software systems [Arts et al., 2006, Hughes and Bolinder, 2011, Hughes et al., 2010], and is widely used among functional programmers.

But property-based testing is not just a way of automating test case generation. A property is a *testable specification* that has a logical meaning! For example, the meaning of `prop_sort` above is $\forall$`xs.sorted(sort(xs))`. QuickCheck tests that this formula is always true, but we might just as well model-check it [Fredlund and Svensson, 2007, 2010] or prove it correct. Despite having logical meaning, QuickCheck properties are ordinary Haskell functions: the programmer doesn't need to learn a special program logic. QuickCheck properties are formal specifications—*that programmers actually write*!

This thesis We want programmers to reason about their programs: to think about what properties their program should satisfy, to write them down, to make sure that they hold either by testing or proving, and to use those properties when programming. Programmers need tools to help with this.

We are convinced that reasoning and specification should be based on *testable properties*: most programs are wrong, so the first purpose of verification must be to find bugs. We think that testable specifications give much of the benefit of formal reasoning. They increase confidence that the program is correct, and improve program understanding. There is no room for woolly thinking, since testing uncovers false properties. What's more, functional programmers already write testable specifications, in the form of QuickCheck properties!

In the end, we would like a wide variety of tools centred around testable properties. We want to be able to discover them, test them, prove them, model check them, symbolically execute them, and so on. We would start by writing a specification, and having QuickCheck test it. When the program seems to be bug-free, we can move on to more powerful techniques like proof, if we want, without the obstacle of having to write new properties. We could mix and match techniques, proving some parts of the program correct while testing others, and because everything is tested, we would rarely try to prove anything

that isn't true. We think this would be a lightweight and compelling approach to verification.

This thesis presents some progress we have made towards these goals:

- QuickSpec, a tool that discovers properties, which helps program understanding and lowers the barrier to specifying a program (paper 1).
- HipSpec, an automatic theorem prover for Haskell programs, which is able to generate and prove the lemmas that are needed for a proof to go through (paper 2).
- Methods for encoding types efficiently in first-order logic, a key ingredient when using first-order provers to reason about programs (papers 3 and 4).
- Tools that use properties to detect and provoke race conditions in concurrent programs (papers 5 and 6).

We have noticed two pervasive themes in our tools and papers:

1. Avoiding side effects makes testing, reasoning and verification much more tractable. Many of our ideas work well because side effects are sparse—in a wilder language, everything would be much harder. For example, it is not obvious how to replicate QuickSpec in an imperative language.
2. Much of what we do works because our property language, QuickCheck properties, is quite weak logically. Most obviously, all of our tools need to test properties, even the theorem prover HipSpec. The fact that HipSpec properties do not use existential quantification allows us to use first-order instead of higher-order reasoning. If we want good tools, we should choose the *least expressive* property language that's powerful enough for us!

QuickCheck as a logic As we mentioned above, QuickCheck properties are quite weak logically. In particular, there is no existential quantification. Is this really enough for general-purpose reasoning? We think so.

If you want to use existential quantification in QuickCheck, you have to be able to compute a witness for the existential quantifier. This is normally possible, either using exhaustive search, or knowledge of the

program you are testing, i.e., white-box testing. The latter makes the property implementation-specific; we can alleviate this by parametrising the property on the expression that computes the witness. This is the price we pay for being able to test all our properties.

Functional programming Many programming language features make reasoning difficult. As an extreme example, saying anything at all about the behaviour of a C program is difficult, because a buffer overflow, or the appearance of undefined behaviour, may break all the assumptions you had about the program.

In imperative languages, the combination of side effects and aliasing is painful. An innocuous operation can have an unexpected side effect, and if we do not know anything about the structure of the heap, many expressions can unexpectedly change their value. Preconditions can get quite complex because they need to restrict aliasing.

In object-oriented languages, overriding means that we do not know the body of a method when we call that method: someone could've overridden it. Thus, unless we specify a precondition and postcondition for that method, or declare it as non-overloadable (`final` in Java), we have no idea what it will do.

Languages that are designed for formal reasoning, like Dafny [Leino, 2010] and SPARK [Barnes, 2003], usually leave out troublesome features. Dafny rules out overloading and ask the user to specify *frames* that say what objects a method may access, to tame the heap, while SPARK simply disallows aliasing altogether.

By contrast, functional programs are easy to reason about. Because there are few side effects, you can reason about small pieces of code in isolation, and informal reasoning is quite likely to work. Specifications are normally short and sweet and free of strange side conditions. Higher-order functions and lazy evaluation allow programs to be broken up that would otherwise be indivisible [Hughes, 1989], so well-written functional programs can be more modular than their imperative counterparts.

All of this means that there is already a culture of reasoning, formal and informal, in functional programming. Many Haskell programmers write down laws they expect to hold, and there is a sizeable group that performs Dijkstra-style program derivations. Many parts of the standard library have been tested with QuickCheck. In short, functional programming is ready for verification!

Paper I: QuickSpec: Formal Specifications for Free!

It is hard to develop a good set of properties from scratch, especially if you are dealing with a program someone else has written. We have written a tool called QuickSpec that takes a side-effect-free module of a functional program and, by testing it, produces a set of *equations* that describe the module. You can use these equations as properties for testing, or just to help you understand the module.

For example, Hughes [1995] describes a library for pretty-printing that includes the following functions:

```
($$), (<>) :: Layout -> Layout -> Layout
nest :: Int -> Layout -> Layout
text :: [Char] -> Layout
```

A `Layout` is roughly a combination of text to be printed and layout information. The `text` function turns any string into a trivial `Layout`; the `nest` function takes any `Layout` and causes it to be indented when it's printed; the `$$` operator combines two `Layouts` vertically, placing one on top of the other, whereas `<>` will put one `Layout` next to the other, on the same line. The pretty-printing library also includes a `sep` operator, which combines two `Layouts` either horizontally or vertically: horizontally if there is enough space to do so, vertically otherwise. This function is really where the power of the library comes from, but we leave it out here for simplicity.

To run QuickSpec, we simply give the list of functions we want to test, in this case the ones above. We also give QuickSpec any auxiliary functions and constants that we think may be useful for specifying the API:

```
0 :: Int
(+) :: Int -> Int -> Int
"" :: [Char]
(++) :: [Char] -> [Char] -> [Char]
```

In the current implementation, we also have to give a collection of variables that the equations may quantify over:

```
i, j, k :: Int
x, y, z :: Elem
d, e, f :: Layout
s, t, u :: [Char]
```

Once we have done this, QuickSpec produces the following equations (and no others), all tested on several hundred randomly-generated test cases:

```
1: nest 0 d == d
2: nest j (nest i d) == nest i (nest j d)
3: d<>nest i e == d<>e
4: nest (i+j) d == nest i (nest j d)
5: (d$$e)$$f == d$$(e$$f)
6: nest i d<>e == nest i (d<>e)
7: (d$$e)<>f == d$$(e<>f)
8: (d<>e)<>f == d<>(e<>f)
9: nest i d$$nest i e == nest i (d$$e)
10: text s<>text t == text (s++t)
11: d<>text "" == d
```

These laws give us some insight into the behaviour of the pretty-printing library. For example, law 3 tells us that horizontally composing two layouts, `d<>e`, ignores the indentation level of `e`—in other words, only the indentation of the *first* thing on each line counts. Law 9 tells us that when indenting a multi-line layout, each line is individually indented. (If only the first line were indented, we would instead see the law `nest i d$$e == nest i (d$$e)`.) Line 10 tells us that the characters in a string are composed horizontally when we use `text`.

In his paper, Hughes gives a list of 11 axioms that characterise the pretty-printing combinators. 10 of these 11 are also found by QuickSpec! (The 11th, which gives conditions when it is possible to push a `<>` inside a `$$`, is a little too complicated for QuickSpec to find.) QuickSpec produces one extra law, number 2. This law is a trivial consequence of number 4, but QuickSpec does not remove it because it considers law 2 to be simpler. QuickSpec can indeed discover useful specifications for nontrivial libraries, something we expand upon in the paper.

Related work in Java Henkel et al. [2007] describe a similar system for Java programs. Their system also uses testing to generate equations that seem to hold for a collection of functions. This is a bit surprising, because imperative programs are normally specified using pre- and postconditions, rather than equations.

The reason is that Henkel et al. only deal with a small fragment of Java. The terms in their equations consist of a chain of method

calls applied to a single object: `obj.f(…).g(…)`. The arguments to the method calls must be atomic—variables or constants. This restriction neatly rules out aliasing, since each term will only mention one object. Because of this, they can model their stateful methods by pure functions taking the "old" object to the "new" object. This allows them to safely use equational reasoning.

Unfortunately, this restriction severely limits the scope of their tool. They are not able to deal with nested method calls `f(g(x))`, and their terms can only mention a single object. This means that they do not support binary operators: if testing, say, a "set" class, they can find laws about `insert` and `delete` but not `union`. Their tool is designed expressly for finding laws about data structures, where these restrictions might not be too onerous; QuickSpec is general-purpose.

Side effects are trouble We are not trying to rubbish Henkel et al.'s tool. Rather, they have an impossible job because equational reasoning does not mix well with side effects, and a Java program consists of nothing *but* side effects chained in the right order. By contrast, large parts of functional programs are side-effect-free—even in an impure language—so that we are able to completely disallow side effects in our equations and still have a convincing tool.

Paper II: HipSpec: Automating Inductive Proofs using Theory Exploration

To prove a purely functional program correct—at least if it doesn't rely on infinite values—all you need is equational reasoning and induction. This makes proving much more accessible than in an imperative language, especially given that many functional programs have simple specifications.

Provers like Waldmeister [Hillenbrand et al., 1997], Vampire [Riazanov and Voronkov, 2002], E [Schulz, 2004], Z3 [de Moura and Bjørner, 2008], and so on, can easily perform equational reasoning. The Haskell inductive prover Hip [Rosén, 2012] builds on these provers to perform inductive proofs, by generating equational proof obligations and sending them to an equational prover. For example, to prove `xs++[] = xs`, Hip will choose to do induction on `xs`, and it will ask the equational prover to prove `[]++[] = []` (the base case) and to prove `(x:xs)++[] = x:xs` assuming that `xs++[] = xs` (the step case). Both proofs will go through and Hip will report `xs++[] = xs` as proved.

Then why are automated proofs of program correctness hard? The trouble lies with induction. To prove something by induction, you may need to generalise the induction hypothesis, or first prove some auxiliary lemmas, something which is never necessary for pure equational reasoning because of cut-elimination. Finding the right lemmas to prove is the problem. For example, if we try to prove that `reverse (reverse xs) = xs` by structural induction on `xs`, the step case goes:

```
  reverse (reverse (x:xs))
=   { definition of reverse }
  reverse (reverse xs ++ [x])
```

and we are stuck. One way to make the proof go through is to prove the generalisation `reverse (reverse xs ++ ys) = reverse ys ++ xs`, and now the step case goes

```
  reverse (reverse (x:xs) ++ ys)
=   { definition of reverse }
  reverse ((reverse xs ++ [x]) ++ ys)
=   { associativity of ++; definition of ++ }
  reverse (reverse xs ++ (x:ys))
=   { induction hypothesis }
  reverse (x:ys) ++ xs
=   { definition of reverse }
  (reverse ys ++ [x]) ++ xs
=   { associativity of ++; definition of ++ }
  reverse ys ++ (x:xs)
```

Thus, in order to prove a simple property, `reverse (reverse xs) = xs`, we needed to both generalise it and first prove the associativity of `++`.[1]

Lemmas such as the associativity of `++` form part of the background theory of a problem—laws that are essential in many proofs. When proving a theorem, it is hard to automatically invent useful background lemmas. Our idea with HipSpec is simple: we *already have* a tool that will discover lemmas for us, namely QuickSpec! What we can do is run QuickSpec to discover some background lemmas, and prove as many of them as possible using Hip. Once we have done that, we take all we managed to prove and use them as background lemmas in the main proof. The tricky part is what order to try to prove the discovered

[1] There is in fact another generalisation, `reverse (ys ++ [x]) = x:reverse ys`, that works and does not use the associativity of `++` [Bird, 1998], but often there is not.

lemmas in; we have a few simple heuristics, detailed in the paper, that seem to work quite well.

Despite its simplicity, HipSpec seems to work rather well. It can prove the following property about the `rotate` function, which has been identified as a challenge "beyond the current state-of-the-art for automated reasoning systems" [Bundy et al., 2005], in under 20 seconds:

```
prop_rotate xs = rotate (length xs) xs == xs
  where
    rotate 0     xs     = xs
    rotate (n+1) []     = []
    rotate (n+1) (x:xs) = rotate n (xs ++ [x])
```

As far as we know, no other theorem prover can prove `prop_rotate` fully automatically. HipSpec first proves the basic list laws `xs++[] = xs` and `xs++(ys++zs) = (xs++ys)++zs`, and after proving a few other properties, hits upon the required generalisation,

```
rotate (length xs) (xs++ys) = ys++xs,
```

which goes through by induction with the help of the list laws.

HipSpec's approach to inductive proof is unorthodox. All other automated induction systems that we know of use top-down lemma discovery, if they do lemma discovery at all: first the system tries to prove the theorem directly, and if that fails, it uses the failed proof attempt to come up with a lemma to prove. Systems that work in this way include ACL2 [Chamarthi et al., 2011], CLAM [Ireland and Bundy, 1995], IsaPlanner [Dixon and Fleuriot, 2004] and Zeno [Sonnex et al., 2011]. Much work in top-down lemma discovery centres on rippling, a rewriting-based technique that guarantees that all proof attempts will terminate, either by proving the property or getting stuck. *Proof critics* can look at a stuck rippling proof and suggest a lemma, for example by generalising the property to be proved.

HipSpec, on the other hand, works bottom-up. It does not even look at the property it is going to prove. Instead, it proves as many interesting lemmas as it can, in the hope that it will end up with a rich enough background theory to prove the main property.

Theory exploration systems such as IsaCoSy [Johansson et al., 2011] and IsaScheme [Montano-Rivas et al., 2012] can generate and prove interesting lemmas automatically, much like HipSpec does, but neither can be used as a theorem prover in the way HipSpec can. They are also much slower than HipSpec. One reason for this is that QuickSpec

does not directly generate equations; rather, it generates a lot of terms, uses testing to divide them into equivalence classes and then extracts equations from the equivalence relation. Thus it can find all equations over n terms in time proportional to n. IsaCoSy and IsaScheme generate and test individual equations instead, because they support arbitrary formulas; this means that it takes time proportional to n^2 to find all equations over n terms. Sticking to an *inexpressive* logic here—namely, equations with free variables—makes the problem much more tractable!

Top-down and bottom-up lemma discovery seem to complement each other. Top-down lemma discovery struggles to discover basic principles like the associativity of ++, but is good at forcing proofs through when the right background theory is present. HipSpec is excellent at finding background lemmas such as the associativity of ++, but it can only discover lemmas up to a certain size. We could imagine using the two in conjunction: HipSpec to find a good background theory, and top-down lemma discovery to attack the particular property we want to prove, once we have a good background theory.

The version of HipSpec we present here only targets a small subset of Haskell. In particular, we require all functions to terminate, and the onus is on the user to check that. The next step is to extend HipSpec to full Haskell, including non-termination and bottom, and we are currently in the middle of that. The main obstacle is that some equations will only hold for total values; we overcome that by generating the correct totality preconditions for all equations, and proving functions total so that we can use those equations.

Another problem is that HipSpec does not discover conditional lemmas. When testing merge sort, for example, we would want to discover the property $\forall xs, ys.\, \mathtt{sorted}(xs) \wedge \mathtt{sorted}(ys) \implies \mathtt{sorted}(\mathtt{merge}(xs, ys))$. Since QuickSpec only deals with equations, what we want is a *conditional equation* that states that $\mathtt{sorted}(\mathtt{merge}(xs, ys)) = \mathtt{true}$ only when xs and ys are sorted. QuickSpec does not yet support conditional equations; we plan to add support, and when we do we hope to be able to prove much more!

Can we reason about functional programs with first-order logic? HipSpec is based on first-order logic, but functional programming languages have higher-order functions. Do we not need higher-order logic to reason about them?

The answer is no. We can simply defunctionalise the program [Reynolds, 1972] and then we will have a first-order program.

We only need higher-order logic if our property existentially quantifies over a function. Lucky for us that our property language doesn't have existential quantification!

Paper III: Sort it out with monotonicity

Historically, most first-order provers have not supported types. But many problems are naturally expressed using types, including most problems in program verification. In this paper, we set out to find an efficient way to encode types in first-order logic.

Logicians have known since the dawn of time how to do this, and many first-order logic books tell you how. The idea is to encode types with *typing predicates*. For each type T in your formula, introduce a unary predicate p_T, which means "has type T", and then apply the following transformation: whenever your formula has a universal quantification $\forall x : T. (\ldots)$, replace it with $\forall x. p_T(x) \to (\ldots)$. (You also have to add some axioms about p_T, which we will ignore.)

The trouble is that this encoding generates a lot of gunk. We will introduce one p_T literal for each quantifier in the formula, which could well double the formula's size. The prover will spend a lot of time proving that different terms have type T rather than doing useful work.

What we would like to do is just *erase* the types. This is not always sound. For example, the formula $(\forall x, y : T. x = y) \wedge (\exists x, y : U. x \neq y)$ is satisfiable—it states that the type T has only one element but the type U has two. If we erase the types, we get $(\forall x, y. x = y) \wedge (\exists x, y. x \neq y)$, which is contradictory.

We have discovered a condition when type erasure is sound. Our condition is called *monotonicity*. A type in a formula is monotonic if, given a model of that formula, you can extend the domain of the type with one extra element and still have a model. The type T above is not monotonic, because it must have only one element: we cannot add a second element to the domain of T. We prove that you can safely erase monotonic types (you will have to read the paper to find out why). In the example above, it is safe to erase U but we will need a typing predicate for T.

We have also designed calculi that detect monotonic types. Many types are monotonic, including any infinite type and any type that we don't use positive equality over (the second implies that U is monotonic above). We have written a tool called Monotonox that implements our efficient type encodings.

Monotonicity also gives us a lovely way of reasoning about type encodings. Suppose we want to show that our encoding of the formula above is correct. We split the encoding up into two stages: first we add a typing predicate for T, but *without erasing any types*, then we erase the types. (Again, we omit the axioms about p_T in the figure below.)

$$(\forall x, y : \mathsf{T}.\, x = y) \wedge (\exists x, y : \mathsf{U}.\, x \neq y)$$

$$\Downarrow$$

$$(\forall x, y : \mathsf{T}.\, \mathsf{p}_\mathsf{T}(x) \wedge \mathsf{p}_\mathsf{T}(y) \rightarrow x = y) \wedge (\exists x, y : \mathsf{U}.\, x \neq y)$$

$$\Downarrow$$

$$(\forall x, y.\, \mathsf{p}_\mathsf{T}(x) \wedge \mathsf{p}_\mathsf{T}(y) \rightarrow x = y) \wedge (\exists x, y.\, x \neq y)$$

From the arguments above, we know that the middle formula is monotonic. Hence we can safely erase the types in the middle formula to obtain the bottom formula! All we need to do is show that the top and middle formulas have the same meaning, i.e., from a model of one we can get a model of the other; this is much easier because we have preserved the types. For example, if we have a model of the top formula, we can obtain a model of the middle formula by letting $\mathsf{p}_\mathsf{T}(x)$ be true everywhere. The other direction is also straightforward.

The resulting proof—which is not fully developed until the next paper—is much cleaner than the traditional textbook proofs, which add the typing predicates and erase the types at the same time. It is not just a mathematical curiosity, either: it makes it clear why each aspect of the encoding is needed. For example, the typing predicates themselves ensure monotonicity, while the typing axioms happen to be needed to go from a model of the middle formula to a model of the top formula. This in turn helped us design the more efficient encodings in the next paper: we can easily see if anything in our encodings is unnecessary and if a proposed encoding is sound.

The notion of monotonicity was invented by Blanchette and Krauss [2011], who used it to help with model finding in higher-order logic. We were the first to transfer monotonicity to first-order logic, and to use it for erasing types. There seems to be very little work about encoding types before ours: most people either used the inefficient, traditional type encodings, or erased types and hoped for the best. Now we can leave it to Monotonox!

Paper IV: Encoding Monomorphic and Polymorphic Types

This paper solves two limitations of the last paper:

1. It only applies to monomorphic problems.
2. The type encodings are not as efficient as they could be.

We will explain the second point first. The last paper allows us to leave out typing predicates for monotonic types, but for non-monotonic types we use the same inefficient scheme as before. In fact, the monotonicity calculus detects *dangerous occurrences* of variables that cause their types to perhaps not be monotonic, and it is only the dangerous variables that we need guard with a typing predicate! This refinement makes our type encodings extremely slim: normally almost everything is erased. Our proof sketch from the last section goes through entirely unchanged; in fact, looking at this proof is how we discovered the refinement in the first place.

This paper also extends our type encodings to polymorphism. Polymorphism in first-order logic is a funny beast because it is not parametric; for example, we may define a nullary predicate $p\,\langle\alpha\rangle$ (where α is a type variable), and give it the axioms $p\,\langle T\rangle \wedge \neg p\,\langle U\rangle$. This is satisfiable, but erasing everything yields $p \wedge \neg p$, which is contradictory. Thus our encoding needs to add *type arguments*. Our p would become a unary predicate $p(A)$ (where A is a *term* variable), with axioms $p(t) \wedge \neg p(u)$.

Even without strange predicates like p, we still need to encode type arguments. Suppose we model combinatory logic, by defining functions app, s and k with the appropriate axioms, and giving them the usual polymorphic types. We know that we cannot define a fixpoint combinator, because polymorphic lambda calculus is strongly normalising. But if we erase the types without encoding any arguments, we get the *untyped* s and k combinators, from which we can define a fixpoint combinator!

We lift as much as possible of the theory of the last paper to polymorphism. In particular, we can define monotonicity for a polymorphic problem. Because of the problem with type arguments, type erasure is not always sound for monotonic problems, but *encoding type arguments* followed by type erasure is. With that proved, we can easily lift our type encodings to polymorphism. This unfortunately results in many type arguments appearing in the untyped problem; we define alternative encodings that encode fewer type arguments at the cost of guarding more variables.

Isabelle's Sledgehammer prover [Blanchette et al., 2011], which translates higher-order logic problems to first-order logic, uses our techniques for encoding types. We used Sledgehammer to benchmark our encodings on a variety of provers. For monomorphic problems, our encodings were very efficient, and even beat some provers' native type support! For polymorphic problems, it was best to monomorphise the problem using a heuristic and then apply the monomorphic encodings; the true polymorphic encodings lagged a bit behind. Unfortunately, it seems that encoding polymorphism has some necessary overhead.

Paper V: Finding Race Conditions in Erlang with QuickCheck and PULSE

Functional programs are not immune from race conditions: while there are no data races, there can still be races between two clients that talk to the same server, for example.

Finding race conditions with a hand-written test suite is hopeless, for two reasons:

1. Concurrent programs tend to be highly non-deterministic. Thus a single test case may have several plausible correct outcomes, depending on what order everything happened to execute in. The larger the test case, the more possible outcomes, and there is often no obvious way to systematically enumerate those outcomes. Therefore, it can be hard even to say if a test passed or failed.

2. Race conditions are famously hard to reproduce, and may reveal themselves only under very particular timing conditions. In order for a test case to provoke such a race condition, you often need to "massage" the test case by inserting carefully-chosen delays. Such test cases depend heavily on the exact timing behaviour of the code under test and may become ineffective if you even make the code a little faster or slower—hardly a basis for a good test suite!

   When debugging a race condition, you might want to add print statements to see what is going on—but that often has the effect of skewing the timing and making the race condition vanish again!

Testing concurrent programs without tool support is a nightmare. We present an approach that uses QuickCheck properties to find race conditions. Our approach has two parts, `eqc_par_statem` (short for "Er-

lang QuickCheck parallel state machine") to detect bad behaviour and PULSE to provoke bad behaviour.

eqc_par_statem The idea behind `eqc_par_statem` is to test for one particular property of a concurrent API; namely, that all the functions of that API behave atomically. This is by no means the *only* property we might want to test of a concurrent API, but is normally a desirable property. After all, a concurrent API whose operations *don't* behave atomically will tend to drive its users mad.

How do we test for atomicity? First, the user must supply a *sequential* specification of the API, giving preconditions, postconditions etc. for each operation. Erlang QuickCheck already supports these specifications to help with testing sequential imperative code [Arts et al., 2006].

Using the sequential specification we generate processes that invoke the API. In fact we generate a pair of parallel processes; each consists of a sequence of API commands. Then we run the two processes in parallel and observe the results. Finally, we have to check if in this *particular* run of the system, the API functions behaved atomically.

We do this by trying to *linearise* [Lamport, 1979] the API calls that were made: we search for an interleaving of the two command sequences that would give the same results that we actually observed. In other words, we work out if the system behaved *as if* only one command was running at a time. If there is no such interleaving, then the only possible explanation is that the two processes interfered with each other, and we report an error.

PULSE The second component of our approach is PULSE, a replacement for the standard Erlang scheduler.

The standard Erlang scheduler suffers from two problems when testing concurrent code:

- It leads to non-deterministic behaviour: running the same test several times may result in a different outcome each time, because the order that processes run in is partly a result of chance. This leads to unrepeatable test cases.

- Paradoxically, it is also *too deterministic*: the Erlang scheduler preempts processes at regular intervals, so that each test case will have a similar effect each time we run it. This might result in the system behaving differently when idle and when under load, for

example. Thus a property we test may be *false* but might never be *falsified* if we do not test it in exactly the right circumstances.

PULSE takes over scheduling decisions from the standard Erlang scheduler. Whenever there is a choice to be made—which process should execute next, for example—it chooses randomly. This exposes as wide a range of behaviour as possible from the program under test.

PULSE also *records* the choices it made into a log. If we want to repeat a test case, we can give the log to PULSE and it will make the same scheduling choices again, thus giving us repeatability. We can also turn this log into a graphical trace of the system's behaviour, to help with debugging.

Case study We applied our tools to a piece of industrial Erlang code with a mysterious race condition, which is described in the paper. We indeed found the race condition; unfortunately, it turned out to be a subtle design fault and impossible to fix without redesigning the API! This shows the importance of finding bugs early.

Related work Imperative programmers are plagued by race conditions too, so naturally similar tools to PULSE exist. For .NET there is Chess [Musuvathi et al., 2008] and for Java there is RaceFuzzer [Sen, 2008].

When comparing PULSE to these tools, the most surprising thing is how much simpler our approach is. We speculate that this is due to the fact that Java and .NET use shared memory concurrency. Because of this, almost any line of code can potentially participate in a race: every operation has an effect that can be observed by other processes, making the problem of finding the real race conditions harder because there are so many irrelevant scheduling decisions. By contrast, Erlang uses message passing concurrency and only provides shared state in the form of a mutable map library, `ets`. PULSE only needs to preempt a process when it sends a message or performs a genuine side effect, rather than on every variable access, so there are far fewer ways to schedule any program and PULSE has an easier time finding the schedules that provoke bugs. We make the problem easier by working in a setting where side effects are only used when actually useful.

The other main difference between our work and that for Java and .NET is that they don't have any equivalent of `eqc_par_statem` for atomicity testing. Chess requires the user to state a property they want to hold; for example, that the system must never deadlock. RaceFuzzer checks that there are no data races, where one thread accesses a variable

at the same time as another thread is writing to it. However, since this is a very low-level property, it sometimes finds harmless data races.

Paper VI: Accelerating Race Condition Detection through Procrastination

PULSE makes scheduling decisions totally at random. This works surprisingly well for many Erlang programs, because there are not too many decisions to be made. However, it is not perfect. In particular, we are unlikely to delay an event for a long time: since at each step we pick an event uniformly at random, the probability of leaving a event for n steps goes down exponentially in n.

We could tweak PULSE's probability distribution, for example making it less likely to pick an event if we have already delayed it for a long time, but whatever distribution we pick there will be some unlikely schedules. Instead, inspired by Sen [2008], we decided to make a tool that would analyse PULSE's schedules and then tweak them to provoke race conditions.

The idea is to look at the schedule and find two events that might interfere, in the sense that reversing the order of the events might provoke a different behaviour. For example, two message sends to the same process interfere. We re-run PULSE, but reverse the order of the two events by steadfastly ignoring the first event until after we have performed the second. We call this technique *procrastination*.

Procrastination has many settings we can tweak. For example, if we have *three* message sends to the same process, should we try them in every possible order? Or, if there are two *pairs* of interfering message sends, should we try reversing both pairs at the same time? The obvious answer to both questions is yes, but procrastination slows down testing: we should not overdo it.

We also tried to fix another problem with PULSE. Once QuickCheck finds a counterexample, it enters a *shrinking* phase, where it tries to simplify the counterexample as much as possible. If we are dealing with a race condition, shrinking will not work well, because when QuickCheck tests a simplified counterexample it will often work by chance. Our solution was simply to ask PULSE to use the same schedule when testing the simplified counterexample that it did on the original counterexample.

We repeated the `proc_reg` example with procrastination, with mixed results. Keeping the schedule while shrinking was a success, and we managed to get the minimal counterexample much more often than

before. We could provoke the race condition more often with procrastination, but only if we used the right settings; with the wrong settings, it was slower.

For procrastination to be more widely useful, it probably needs to be fully automatic, i.e., we need to decide automatically when it might be useful to procrastinate a particular event.

My contributions to the papers

Paper I We came up with the ideas behind QuickSpec jointly. I was responsible for the depth optimisation, the older, rewriting-based pruning algorithm, the instantiation choice in the current pruning algorithm, the definition detection and the current implementation. We wrote the paper jointly.

Paper II Dan Rosén and I wrote the initial implementation together; since then he has done most of the implementation work, with me adapting QuickSpec for use in HipSpec. I was responsible for many of the heuristics in choosing the next equation to prove. We all jointly wrote the paper.

Paper III It was Koen Claessen's idea to use monotonicity to detect when type erasure is sound. Ann Lillieström and I designed the two calculi together and implemented the tool. We all jointly wrote the paper.

Paper IV Roughly speaking, I was responsible for encoding non-monotonic types and Jasmin Blanchette for encoding type arguments. Andrei Popescu was in charge of the Isabelle formalisation and Sascha Böhme contributed the heuristic monomorphisation.

Paper V Ulf Wiger provided the industrial example; the rest of us wrote and experimented with PULSE together, mostly in joint hacking sessions. I was solely responsible for the instrumentation. I was not involved in the design of `eqc_par_statem`. We wrote the paper together.

Paper VI John Hughes and I worked out the details of procrastination together on a plane journey; everything else all of us shared roughly equally.

1

QuickSpec: Formal Specifications for Free!

This paper was originally published at TAP 2010 in Malaga, under the title "QuickSpec: Guessing Formal Specifications using Testing". This is an extended version of the published paper.

Chapter 1

QuickSpec: Formal Specifications for Free!

Koen Claessen John Hughes
Nicholas Smallbone

Abstract

We present QuickSpec, a tool that automatically generates algebraic specifications for sets of pure functions. The tool is based on testing, rather than static analysis or theorem proving. The main challenge QuickSpec faces is to keep the number of generated equations to a minimum while maintaining completeness. We demonstrate how QuickSpec can improve one's understanding of a program module by exploring the laws that are generated using two case studies: a heap library for Haskell and a fixed-point arithmetic library for Erlang.

1 Introduction

Understanding code is hard. But it is vital to understand what code does in order to determine its correctness.

One way to understand code better is to write down one's expectations of the code as formal specifications, which can be tested for compliance, by using a property-based testing tool. Our earlier work on the random testing tool QuickCheck [Claessen and Hughes, 2000] follows this direction. However, coming up with formal specifications is difficult, especially for untrained programmers. Moreover, it is easy to forget to specify certain properties.

In this paper, we aim to aid programmers with this problem. We propose an automatic method that, given a list of function names and their object code, uses testing to come up with a set of *algebraic equations* that seem to hold for those functions. Such a list can be useful in several ways. Firstly, it can serve as a basis for documentation of the code. Secondly, the programmer might gain new insights by discovering new laws about the code. Thirdly, some laws that one expects might be missing (or some laws might be more specific than expected), which points to to a possible mistake in the design or implementation of the code.

Since we use testing, our method is potentially unsound, meaning some equations in the list might not hold: the quality of the generated equations is only as good as the quality of the used test data, so we have to be careful. Nonetheless, we still think our method is useful. However, our method is still complete in a precise sense: although there is a limit on the complexity of the expressions that occur in the equations, any syntactically valid equation that actually holds for the code can be derived from the set of equations that QuickSpec generates.

Our method has been implemented for the functional languages Haskell and Erlang in a tool called QuickSpec. At the moment, QuickSpec only works for purely functional code, i.e. no side effects. (Adapting it to imperative and other side-effecting code is ongoing work.)

1.1 Examples

Let us now show some examples of what QuickSpec can do, by running it on different subsets of the Haskell standard list functions. When we use QuickSpec, we have to specify the functions and variable names which may appear in equations, together with their types. For example, if we generate equations over the list operators

```
(++) :: [Elem] -> [Elem] -> [Elem]     -- list append
(:)  :: Elem -> [Elem] -> [Elem]       -- list cons
[]   :: [Elem]                         -- empty list
```

using variables `x,y,z :: Elem` and `xs,ys,zs :: [Elem]`, then QuickSpec outputs the following list of equations:

```
xs++[] == xs
[]++xs == xs
(xs++ys)++zs == xs++(ys++zs)
(x:xs)++ys == x:(xs++ys)
```

We automatically discover the associativity and unit laws for append (which require induction to prove). These equations happen to comprise a complete characterization of the ++ operator. If we add the list reverse function to the mix, we discover the additional familiar equations

```
reverse [] == []
reverse (reverse xs) == xs
reverse xs++reverse ys == reverse (ys++xs)
reverse (x:[]) == x:[]
```

Again, these laws completely characterize the `reverse` operator. Adding the `sort` function from the standard `List` library, we compute the equations

```
sort [] == []
sort (reverse xs) == sort xs
sort (sort xs) == sort xs
sort (ys++xs) == sort (xs++ys)
sort (x:[]) == x:[]
```

The third equation tells us that `sort` is idempotent, while the second and fourth strongly suggest (but do not imply) that the result of `sort` is independent of the order of its input.

Adding the `usort` function (equivalent to `nub . sort`), which sorts and eliminates duplicates from its result, generates the same equations together with one new one:

```
usort (xs++xs) == usort xs
```

which strongly suggests that the result of `usort` is independent of repetitions in its input.

If we add a `merge` function for ordered lists, then we obtain equations relating `merge` and `sort`:

```
merge [x] (sort xs) == sort (x:xs)
merge (sort xs) (sort ys) == sort (xs++ys)
```

We also obtain other equations about `merge`, such as the somewhat surprising

```
merge (xs++ys) xs == merge xs xs++ys
```

Note that this holds even for *unordered* `xs` and `ys`, but is an artefact of the precise definition of `merge`.

We can deal with higher-order functions as well. Adding the function `map` together with a variable `f :: Elem -> Elem`, we obtain:

```
map f [] == []
map f (reverse xs) == reverse (map f xs)
map f xs++map f ys == map f (xs++ys)
f x:map f xs == map f (x:xs)
```

Because our signature concerns sorting and sorted lists, there may be laws about `map f` that only hold when `f` is monotone. Although QuickSpec does not directly support such *conditional laws*, we can simulate them by adding a new type `Monotonic` of monotone functions. Given an operator that applies a monotone function to its argument...

```
monotonic :: Monotonic -> Elem -> Elem
```

...and a variable `f :: Monotonic`, we obtain

```
map (monotonic f) (sort xs) == sort (map (monotonic f) xs)
merge (map (monotonic f) xs) (map (monotonic f) ys) ==
  map (monotonic f) (merge xs ys)
```

The latter equation is far from obvious, since it applies even to *unordered* `xs` and `ys`.

All of the above uses of QuickSpec only took a fraction of a second to run, and what is shown here is the verbatim output of the tool.

1.2 Queues

The Erlang standard libraries include an abstract datatype of *double-ended queues*, with operations to add and remove elements at the left and the right ends, to join and reverse queues, and so on. The representation is the well-known one using a pair of lists, which gives amortized constant time for many operations [Burton, 1982]. Running QuickSpec on a signature including

```
new() -> queue()
tail(queue()) -> queue()
liat(queue()) -> queue()
reverse(queue()) -> queue()
in(elem(),queue()) -> queue()
```

```
in_r(elem(),queue()) -> queue()
join(queue(), queue()) -> queue()
to_list(queue()) -> list()
```

with the variables

```
X, Y, Z :: elem()
Q, Q2, Q3 :: queue()
```

we obtain equations such as

```
join(new(),Q) == Q
join(Q,new()) == Q
join(join(Q,Q2),Q3) == join(Q,join(Q2,Q3))
to_list(Q) ++ to_list(Q2) == to_list(join(Q,Q2))
```

which tell us that the `join` operator is very well-behaved, and

```
reverse(in(X,Q)) == in_r(X,reverse(Q))
tail(reverse(Q)) == reverse(liat(Q))
```

which relate the insertion and removal operations at each end of the queue to each other in a pleasing way. On the other hand, we also obtain

```
to_list(Q) ++ [X] == to_list(in(X,Q))
[X|to_list(Q)] == to_list(in_r(X,Q))
```

which reveal that if we thought that `in_r` inserted an element at the *right* or the *rear* of the queue, then we were wrong! The `in` function inserts elements on the right, while `in_r` inserts them on the left, of course.

However, some of the generated equations are more intriguing. Consider this one:

```
tail(in_r(Y,Q)) == tail(in_r(X,Q))
```

It is not unexpected that this equation holds—it says that adding an element to the front of a queue, then removing it again, produces a result that does not depend on the element value. What *is* unexpected is that our tool does not report the simpler form:

```
tail(in_r(X,Q)) == Q
```

In fact, the reason that we do not report this simpler equation is that it is not true! One counterexample is `Q` taken to be `in(0,new())`, which evaluates to `{[0],[]}`, but for which the left-hand side of the equation evaluates to `{[],[0]}`. These are two different representations of the "same" queue, but because the representations do differ, then by default QuickSpec considers the equation to be false.

We can also tell QuickSpec to compare the contents of the queues rather than the representation, in which case our equation becomes true and we get a slightly different set of laws.

1.3 Arrays

In 2007, a new library was added to the Erlang distribution supporting purely functional, flexible arrays, indexed from zero [Carlsson and Gudmundsson, 2007]. We applied QuickSpec to subsets of its API. Using the following signature,

```
new() -> array()
get(index(),array()) -> elem()
set(index(),elem(),array()) -> array()
default_element() -> elem()
```

with variables

```
X, Y, Z :: elem()
I, J, K :: index()
A, B, C :: array()
```

we obtained these laws:

```
get(I,new()) == default_element()
get(I,set(I,X,A)) == X
get(I,set(J,default_element(),new())) == default_element()
get(J,set(I,X,new())) == get(I,set(J,X,new()))
set(I,X,set(I,Y,A)) == set(I,X,A)
set(J,X,set(I,X,A)) == set(I,X,set(J,X,A))
```

The `default_element()` is not part of the arrays library: we introduced it and added it to the signature after QuickSpec generated the equation

```
get(I,new()) == get(J,new())
```

Since the result of reading an element from an empty array is constant, we might as well give it a name for use in other equations. When we do so, then the equation just above is replaced by the first one generated.

Some of the equations above are very natural: the second says that writing an element, then reading it, returns the value written; the fifth says that writing to the same index twice is equivalent to just writing the second value. The sixth says that writing the *same* value `X` to two indices can be done in either order—but why can't we swap *any* two writes, as in

```
set(J,Y,set(I,X,A)) =?= set(I,X,set(J,Y,A))
```

The reason is that this equation holds only if `I /= J` (or if `X == Y`, of course)! It would be nice to generate *conditional equations* such as

```
I /= J ==> set(J,Y,set(I,X,A)) == set(I,X,set(J,Y,A)).
```

Presently we have a prototype version of QuickSpec that can generate such laws (and indeed generates the one above), but it is rather preliminary and there are several creases to be ironed out yet. The version of QuickSpec that we discuss in this paper doesn't generate conditional laws.

The fourth equation

```
get(J,set(I,X,new())) == get(I,set(J,X,new()))
```

is a little surprising at first, but it does hold—either both sides are the default element, if `I` and `J` are different, or both sides are `X`, if they are the same.

Finally, the third equation is quite revealing about the implementation:

```
get(I,set(J,default_element(),new())) == default_element()
```

A new array contains the default element at every index; evidently, setting an index explicitly to the default element will not change this, so it is no surprise that the `get` returns this element. The surprise is that the second argument of `get` appears in this complex form. Why is it `set(J,default_element(),new())`, rather than simply `new()`, when both arrays have precisely the same elements? The answer is that *these two arrays have different representations*, even though their elements are the same. That the equation appears in this form tells us, indirectly, that

```
set(J,default_element(),new()) /= new()
```

because if they were equal, then QuickSpec would have simplified the equation. In fact, there is another operation in the API, `reset(I,A)`, which is equivalent to setting index `I` to the default element, and we discover in the same way that

```
reset(J,new()) /= new()
```

`set` and `reset` could have been defined to leave an array unchanged if the element already has the right value—and this could have been a useful optimization, since returning a different representation forces `set` and `unset` to copy part of the array data-structure. Thus this *missing equation* reveals a potentially questionable design decision in the library itself. This is exactly the kind of insight we would like QuickSpec to provide!

The arrays library includes an operation to *fix* the size of an array, after which it can no longer be extended just by referring to a larger index. When we add `fix` to the signature, we discover

```
fix(fix(A)) == fix(A)
get(I,fix(new())) == undefined()
set(I,X,fix(new())) == undefined()
```

Fixing a fixed array does not change it, and if we fix a new array (with a size of zero), then any attempt to get or set an element raises an exception[1].

When we include the array resizing operation, we obtain a number of new laws:

```
get(I,resize(J,new())) == default_element()
get(J,resize(J,A)) == get(I,resize(I,A))
resize(I,resize(I,A)) == resize(I,A)
resize(J,fix(C)) == fix(resize(J,C))
set(I,X,resize(I,new())) == set(I,X,new())
```

The first reveals, indirectly, that

```
resize(J,new()) /= new()
```

[1] We consider all terms which raise an exception to be equal—and `undefined()` is a built-in-to-QuickSpec term that always does so.

which is perhaps not so surprising. The second equation is interesting: it holds because the `I`'th index is just beyond the end of `resize(I,A)`, and so the result is either the default element (if `A` is a flexible array), or `undefined()` (if `A` is fixed). The equation tells us that which result we get depends only on `A`, not on `I` or `J`. It also tells us that the result is not *always* the default element, for example, since if it were, then we would have generated an equation with `default_element()` as its right-hand side.

The reader may care to think about why the third equation specifies only that two resizes *to the same size* are equivalent to one, rather than the more general (but untrue)

```
resize(I,resize(J,A)) =?= resize(I,A)
```

The fourth equation tells us that resizing and fixing an array commute nicely, while the fifth gives us a clue about the reason for the behaviour of `set` discussed earlier: setting an element can clearly affect the *size* of an array, as well as its elements, which is why setting an element to its existing value cannot always be optimized away.

1.4 Main related work

The existing work that is most similar to ours is Henkel et al. [2007]. They describe a tool for discovering algebraic specifications from Java classes using testing, using a similar overall approach as ours (there are however important technical differences discussed in the related work section later in the paper). However, the main difference between our work and theirs is that we generate equations between *nested expressions* consisting of functions and variables whereas they generate equations between Java program fragments that are *sequences of method calls*. The main problem we faced when designing the algorithms behind QuickSpec was taming the explosion of equations generated by operators with structural properties, such as associativity and commutativity, equations that are not even expressible as equations between sequences of method calls (results of previous calls cannot be used as arguments to later ones).

1.5 Contributions

We present a efficient method, based on testing, that automatically computes algebraic equations that seem to hold for a list of specified pure functions. Moreover, using two larger case studies, we show the

usefulness of the method, and present concrete techniques of how to use the method effectively in order to understand programs better.

2 How QuickSpec Works

The input taken by QuickSpec consists of three parts:

- the compiled program,
- a list of functions and variables, together with their types, and
- test data generators for each of the types of which there exists at least one variable.

2.1 The method

The method used by QuickSpec follows four distinct steps:

1. We first generate a (finite) set of terms, called the *universe*, that includes any term that might occur on either side of an equation.
2. We use testing to partition the universe into *equivalence classes*; any two terms in the same equivalence class are considered equal after the testing phase.
3. We generate a list of *equations* from the equivalence classes.
4. We use *pruning* to filter out equations that follow from other equations by equational reasoning.

In the following, we discuss each of these steps in more detail, plus some refinements and optimisations to the basic method. As a running example we take a tiny signature containing the boolean operator `&&` and the constant `false`, as well as boolean variables `x` and `y`.

2.2 The universe

First, we need to pin down what kind of equations QuickSpec should generate. To keep things simple and predictable, we only generate one finite set of terms, the *universe*, and our equations are simply pairs of terms from the universe. Any pair of terms from the universe can form an equation, and both sides of an equation must be members of the universe.

What terms the universe should contain is really up to the user; all we require is that the universe be subterm-closed. The most useful way of generating the universe is letting the user specify a *term depth* (usually 3 or 4), and then simply produce all terms that are not too deep. The terms here consist of the function symbols and variables from the specified API.

The size of the universe is typically 1,000 to 50,000 terms, depending on the application.

To model *exceptions*, our universe also contains one constant at each type, called `undefined`. The behaviour of `undefined` is to throw an exception when you evaluate it. This means that we can also find equations of the form `t == undefined`, which is true if `t` always throws an exception.

In our tiny boolean example, supposing that our maximum term depth is 2, the universe consists of the following terms (we leave out `undefined` for clarity):

```
x      y      false      x&&x       x&&y       x&&false
y&&x   y&&y   y&&false   false&&x   false&&y   false&&false
```

2.3 Equivalence classes

The next step is to gather information about the terms in the universe. This is the only step in the algorithm that uses testing or makes use of the program under test. Here, we need to determine which terms seem to behave the same, and which terms seem to behave differently. In other words, we are computing an equivalence relation over the terms.

Concretely, in order to compute this equivalence relation, we use a refinement process. We represent the equivalence relation as a set of equivalence classes, partitions of the universe. We start by assuming that all terms are equal, and put all terms in the universe into one giant equivalence class. Then, we repeat the following process: We use the test data generators to generate test data for each of the variables occurring in the terms. We then refine each equivalence class into possibly smaller ones, by evaluating all the terms in a class and grouping together the ones that are still equal, splitting the terms that are different. The process is repeated until the equivalence relation seems "stable"; if no split has happened for the last 200 tests. Note that equivalence classes of size 1 are trivial, and can be discarded; these contain a term that is only equal to itself.

The typical number of non-trivial equivalence classes we get after this process lies between 500 and 5,000, again depending on the size of the original universe and the application.

Once these equivalence classes are generated, the testing phase is over, and from here onwards we trust the equivalence relation to be correct.

For our tiny example, the non-trivial equivalence classes are as follows:

```
{x, x&&x}
{y, y&&y}
{false, false&&x, false&&y, x&&false, y&&false, false&&false}
{x&&y, y&&x}
```

2.4 Equations

From the equivalence classes, we can generate a list of equations between terms. We do this by picking one representative term r from the class, and producing the equation $t = r$ for all other terms from the class. So, an equivalence class of size k produces $k - 1$ equations. The equations look nicest if r is the *simplest* element of the equivalence class (according to some simplicity measure based on for example depth and/or size), but which r is chosen has no effect on the completeness of the algorithm.

However, this is not the full story. Taking our tiny boolean example again, we can read off the following equations from the equivalence relation:

```
1. x&&x == x
2. y&&y == y
3. x&&y == y&&x
4. x&&false == false
5. y&&false == false
6. false&&x == false
7. false&&y == false
8. false&&false == false
```

This is certainly *not* what we want to present to the user: there is a mass of redundancy here. Laws 2, 5 and 7 are just renamings of laws 1, 4 and 6; moreover, laws 1, 3 and 4 together imply all the other laws. Our eight equations could be replaced by just three:

```
1. x&&x == x
```

```
2. x&&y == y&&x
3. x&&false == false
```

Whittling down the set of equations is the job of the *pruning* step.

2.5 Pruning

Pruning filters out redundant laws from the set of equations generated above, leaving a smaller set that expresses the same information. That smaller set is what QuickSpec finally shows to the user.

In a real example, the number of equations that are generated by the testing phase can lie between 1,000 and 50,000. A list of even 1,000 equations is absolutely not something that we want to present to the user; the number of equations should be in the tens, not hundreds or thousands. Therefore, the pruning step is crucial.

Which laws are kept and which are discarded by our current implementation of QuickSpec is in some ways an arbitrary choice, but our choice is governed by the following four principles:

1. **Soundness**: we can only remove a law if it can be derived from the remaining laws. In other words, from the set of equations that our pruning algorithm keeps we should be able to derive all of the equations that were deleted.

2. **Conciseness**: we should remove all obvious redundancy, for a suitably chosen definition of redundant.

3. **Implementability**: the method should be implementable and reasonably efficient, bearing in mind that we may have thousands of equations to filter.

4. **Predictability**: the user should be able to draw conclusions from the absence or presence of a particular law, which means that no ad-hoc decisions should be made in the algorithm.

We would like to argue that the choice of which laws to discard must *necessarily* be arbitrary. The "ideal" pruning algorithm might remove all redundant laws, leaving a minimal set of equations from which we can prove all the rest. However, this algorithm *cannot exist*, because:

- Whether an equation is redundant is in general undecidable, so a pruning algorithm will only be able to detect certain classes of redundancy and will therefore print out more equations than we would like.
- There will normally not be one unique minimal set of equations. In our boolean example, we could take out `x&&false == false` and replace it by `false&&x == false` and we would still have a minimal set. Any algorithm must make an arbitrary choice between these two sets.

There is one more point. We do not necessarily *want* to produce an absolutely minimal set of equations, even if we could. Our tool is intended for program understanding, and we might want to keep a redundant equation if it might help understanding. This is another arbitrary choice any pruning algorithm must make. For example, in the domain of boolean expressions, from the following laws...

```
x||true == true
x&&true == x
x&&(y||z) == (x&&y)||(x&&z)
```

...we can actually prove idempotence, `x||x == x`! (Take the third law and substitute `true` for `y` and `z`.) So a truly minimal set of equations for booleans would not include the idempotence law. However, it's quite illuminating and we probably do want to include it.

In other words, we do not want to consider `x||x == x` redundant, even though it is possible to prove it. After long discussions between the authors of the paper on examples like this, it became clear to us that the choice of what is redundant or not is not obvious; here there is another arbitrary choice for our algorithm to make.

Eventually we settled on the following notion of redundancy: we consider an equation redundant if it can be proved from *simpler* equations, where we measure an equation's simplicity in an ad hoc way based on the equation's length, number of variables, etc. For example, we will keep the idempotence law above because you cannot prove it without invoking the more complicated distributivity law. Our justification for this choice is that a simple, general law is likely to be of interest even if it can be proved from more complex principles; in our experience, this seems to be broadly true.

Once we decide to use this "simplicity" metric most of our choices vanish and it becomes clear that our pruning algorithm should work broadly like this:

1. Sort the equations according to "simplicity".
2. Go through each equation in turn, starting with the simplest one:
   - If the equation cannot be proved from the earlier equations in the list, print it out for the user to see: it will not be pruned.
   - If it can be proved, do nothing: it has been pruned.

So now the shape of the pruning algorithm is decided. The problem we are left with is how to decide if a equation is derivable from the earlier equations.

Since derivability is undecidable, we decided to define a decidable and predictable *conservative approximation* of logical implication for equations. The approximation uses a *congruence closure* data-structure, a generalization of a union/find data-structure that maintains a congruence relation[2] over a finite subterm-closed set of terms. Like union/find, it provides two operations: unifying two congruence classes, and testing if two terms are in the same congruence class. Congruence closure is one of the key ingredients in modern SMT-solvers, and we simply reimplemented an efficient modern congruence closure algorithm following Nieuwenhuis and Oliveras [2005].

Congruence closure solves the *ground equation problem*: it can decide if a set of equations implies another equation, but only if the equations contain no variables. It goes like this: start with an empty congruence relation. For each ground equation $t = u$ that you want to assume, unify t's and u's congruence class. Then to find out if $a = b$ follows from your assumptions, simply check if a and b lie in the same congruence class. Note that the ground equation problem is decidable, and efficiently solvable: the congruence closure algorithm gives the correct answer, and very quickly.

We can try to apply the same technique to check for logical implication even when the equations are not ground. However, there is an extra inference rule we have to take care of, namely that any instance of a valid equation is valid: if $t = u$ then $t\sigma = u\sigma$ (where σ is any substitution). Congruence closure does not capture this rule. Instead, we *approximate* the rule: whenever we want to assume a non-ground equation $t = u$, instead of just unifying t's and u's congruence class, we must generate a large number of instances $t\sigma = u\sigma$ and for each of them we must unify $t\sigma$'s and $u\sigma$'s congruence class.

[2] A congruence relation is an equivalence relation that is also a congruence: if $x \equiv y$ then $C[x] \equiv C[y]$ for all contexts C.

In other words, given a set of equations E to assume and an equation $a = b$ to try to prove, we can proceed as follows:

1. Start with an empty congruence relation.
2. For each equation $t = u$ in E, generate a set of substitutions, and for each substitution σ, unify $t\sigma$'s and $u\sigma$'s congruence class.
3. Finally, to see if $a = b$ is provable from E, just check if a and b lie in the same congruence class.

This procedure is sound but incomplete: if it says $a = b$ is true, then it is; otherwise, $a = b$ is either false or just too difficult to prove.

By plugging this proof procedure into the "broad sketch" of the pruning algorithm from the previous page, we get a pruning algorithm. Doing this naively, we would end up reconstructing the congruence relation for each equation we are trying to prune; that would be wasteful. Instead we can construct the congruence relation incrementally as we go along. The final pruning algorithm looks like this, and is just the algorithm from the previous page specialised to use our above heuristic as the proof method:

1. Start with an empty congruence relation. This relation will represent all knowledge implied by the equations we have printed so far, so that if t and u lie in the same congruence class then $t = u$ is derivable from the printed equations.
2. Sort the equations according to "simplicity".
3. Go through each equation $t = u$ in turn, starting with the simplest one:
   - If t and u lie in the same congruence class, do nothing: the equation can be pruned.
   - Otherwise:
     a) Print the equation out for the user to see: it will not be pruned.
     b) Generate a set of substitutions; for each substitution σ, unify $t\sigma$'s and $u\sigma$'s congruence class. (This means that the congruence relation now "knows" that $t = u$ and we will be able to prune away its consequences.)

The final set of equations that is produced by the pruning algorithm is simply printed out and shown to the user as QuickSpec's output.

2.6 Which instances to generate

One thing we haven't mentioned yet is which instances of each equation $t = u$ we should generate in step 3b of the pruning algorithm—this is a crucial choice that controls the power of the algorithm. Too few instances and the pruner won't be able to prove many laws; too many and the pruner will become slow as we fill the congruence relation with hundreds of thousands of terms.

Originally we generated all instances $t\sigma = u\sigma$ such that both sides of the equation are members of the universe (recall that the universe is the large set of terms from which all equations are built). In practice this means that we would generate all instances up to a particular term depth.

This scheme allows the pruning algorithm to reason freely about the terms in the universe: we can *guarantee* to prune an equation if there is an equational proof of it where all the terms in all of the steps of the proof are in the universe, so the pruner is quite powerful and predictable.

While this scheme works pretty well for most examples, it falls down a bit when we have operators with structural properties, such as associativity. For example, generating properties about the arithmetic operator +, we end up with:

```
1. x+y = y+x
2. y+(x+z) = (z+y)+x
3. (x+y)+(x+z) = (z+y)+(x+x)
```

The third equation can be derived from the first two, but the proof goes through a term `x+(y+(x+z))` that lies outside of the universe, so our original scheme doesn't find the proof.

To fix this we relaxed our instance generation somewhat. Instead of requiring both $t\sigma$ and $u\sigma$ to be in the universe, we require only *one* of them to be in the universe. Formally, we look at t and generate all substitutions σ such that $t\sigma$ is in the universe; then we look at u and generate all substitutions σ such that $u\sigma$ is in the universe; then we apply each of those substitutions to $t = u$ to get an instance, which we add to the congruence relation.

By relaxing the instance generation, we allow the algorithm to reason about terms that lie *outside* the universe, in a limited way. While the original scheme allows the pruner to find all equational proofs where all

intermediate terms are in the universe,[3] the relaxed scheme also allows us to "jump out" of the universe for one proof step: in our proofs we are allowed to apply an equation in a way that takes us to a term *outside* the universe, provided that we follow it immediately with another proof step that takes us back into the universe. Our pruner is *guaranteed* to prune a law if there is a proof of it fulfilling this restriction.

Adding the modification we just described to the algorithm, the last equation is also pruned away. The modification does not noticeably slow down QuickSpec.

An example

We now demonstrate our pruning algorithm on the running booleans example. To make it more interesting we add another term to the universe, `false&&(x&&false)`. Our pruning algorithm sorts the initial set of equations by simplicity, giving the following:

```
1. x&&x == x
2. y&&y == y
3. x&&y == y&&x
4. x&&false == false
5. y&&false == false
6. false&&x == false
7. false&&y == false
8. false&&false == false
9. false&&(x&&false) == false.
```

We start with an empty congruence relation $\equiv$ in which every term is in its own congruence class:

```
{x} {y} {false} {x&&x} {y&&y} {x&&y} {y&&x}
{x&&false} {y&&false} {false&&x} {false&&y}
{false&&false} {false&&(x&&false)}
```

Starting from equation 1, we see that `x&&x` and `x` are in different congruence classes (we can't prove them equal) so we print out `x&&x == x` as an equation. We add its instances to the congruence-closure by unifying

[3] Although the pruning algorithm does not exactly search for *proofs*, it is useful to characterise the pruning algorithm's power by what proofs it can find: "if it is possible to prove the equation using a proof of *this form* then the equation will be pruned"

`x&&x` with `x`, `y&&y` with `y` and `false&&false` with `false`.[4] The congruence relation now looks like this:

```
{x, x&&x} {y, y&&y} {false, false&&false} {x&&y} {y&&x}
{x&&false} {y&&false} {false&&x} {false&&y}
{false&&(x&&false)}
```

Coming to equation 2, we see that `y&&y` and `y` *are* in the same congruence class, so we prune away the equation: we can prove it from the previous equation. Equation 3 isn't provable, so we print it out and add some instances to the congruence closure data structure—`x&&y` ≡ `y&&x`, `x&&false` ≡ `false&&x`, `y&&false` ≡ `false&&y`. Now the congruence relation is as follows:

```
{x, x&&x} {y, y&&y}, {false, false&&false} {x&&y, y&&x}
{x&&false, false&&x} {y&&false, false&&y}
{false&&(x&&false)}
```

Equation 4, `x&&false == false`, isn't provable either, so we print it out. We generate some instances as usual—`x&&false` ≡ `false`, `y&&false` ≡ `false`, `false&&false` ≡ `false`. This results in the following congruence relation:

```
{x, x&&x} {y, y&&y}
{false, x&&false, false&&x, y&&false, false&&y,
 false&&false, false&&(x&&false)}
{x&&y, y&&x}
```

Notice that `false&&(x&&false)` is now in the same congruence class as `false`, even though we never unified it with anything. This is the extra feature that congruence closure provides over union/find—since we told it that `x&&false` ≡ `false` and `false&&false` ≡ `false`, it deduces by itself that `false&&(x&&false)` ≡ `false&&false` ≡ `false`.

Looking at the remainder of our equations, both sides are always in the same congruence class. So all the remaining equations are pruned away and the final set of equations produced by QuickSpec is

```
1. x&&x == x
2. x&&y == y&&x
3. x&&false == false
```

as we desired.

[4] According to our relaxed instance-generation scheme from above, we should generate even more instances because `x` can range over any term in the universe, but we ignore this for the present example.

2.7 Alternative pruning methods that Don't Work

The above pruning algorithm may seem overly sophisticated (although it is rather short and sweet in practice because the congruence closure algorithm does all the hard work). Wouldn't a simpler algorithm be enough? We claim that the answer is no: in this section we present two such simpler pruning algorithms, which we used in earlier versions of QuickSpec; both algorithms fail to remove many real-life redundant laws.

We present these failed algorithms in the hope that they illustrate how well-behaved our current pruning algorithm is.

Instance-based pruning

In our first preliminary experiments with QuickSpec, we used the following pruning algorithm: simply delete an equation if it's an instance of another one.

For our boolean example this leads to the following set of laws:

```
1. x&&x == x
2. x&&y == y&&x
3. x&&false == false
4. false&&x == false
```

which is not satisfactory: laws 3 and 4 state the same thing modulo commutativity. However, to prove law 4 you need to use laws 3 and 2 together, and this simplistic pruning method is not able to do that: it will only prune a law if it's an instance of *one* other law. In all real examples this simplistic algorithm is hopelessly inadequate.

Rewriting-based pruning

We next observed that we ought to be able to delete the equation `t == u` (`u` being `t`'s representative), if we can use the other equations as *rewrite rules* to simplify `t` to `u`, where every rewrite rule must simplify the term. In fact, because our universe is closed under "simplicity", if we can simplify `t` at all, we can delete the equation `t == u`. Thus we can efficiently implement this idea.

This approach works just fine on our booleans example. The problematic equation for instance-based pruning was `false&&x == false`; we can use the commutativity of `&&` to rewrite `false&&x` to `x&&false`, and the law `x&&false == false` to rewrite that to `false`.

Why it doesn't work So why don't we use this pruning method any more? It seems promising at first glance and is quite easy to implement. Indeed, we used it in QuickSpec for quite a long time, and it works well on some kinds of examples, such as lists. However, it is unable to apply a rewrite step "backwards", starting from a simpler term to get to a more complicated term; this is unfortunately often necessary. In particular:

1. Commutativity laws such as `x&&y == y&&x` have no natural orientation. So wemay be allowed to rewrite `t&&u` to `u&&t` or it may not; this is determined arbitrarily depending on which term our equation order considers simpler. If the proof of an equation uses commutativity in the wrong direction, the rewriting-based pruner won't be able to delete the equation.

2. If we have in our signature one operator that distributes over another, for example `x&&(y||z) == (x&&y)||(x&&z)`, the pruner will only be able to apply this law from right to left (contracting a term). However, there are many laws that we can only prove by using distributivity to expand a term followed by simplifying it. The rewriting-based pruner will not be able to filter out those laws.

Does this matter in practice? To answer, we list just a small selection of the laws produced by the rewriting-based pruning algorithm when we ran QuickSpec on a sets example. The signature contains four operations, `new` which returns the empty set, `add_element` which inserts one element into a set, and the usual set operators `union` and `intersection`. The rewriting-based pruning algorithm printed out these laws, along with dozens of others:

```
union(S,S) == S
union(S,new()) == S
union(T,S) == union(S,T)
union(U,union(S,T)) == union(S,union(T,U))
union(S,union(S,T)) == union(S,T)
union(union(S,T),union(T,U)) == union(S,union(T,U))
union(union(S,U),union(T,V)) == union(union(S,T),union(U,V))
add_element(X,add_element(X,S)) == add_element(X,S)
add_element(Y,add_element(X,S)) ==
  add_element(X,add_element(Y,S))
union(S,add_element(X,T)) == add_element(X,union(S,T))
```

```
union(add_element(X,S),add_element(X,T)) ==
  add_element(X,union(S,T))
union(add_element(X,S),add_element(Y,S)) ==
  add_element(X,add_element(Y,S))
union(add_element(X,S),union(S,T)) ==
  add_element(X,union(S,T))
union(add_element(X,T),add_element(Y,S)) ==
  union(add_element(X,S),add_element(Y,T))
union(add_element(X,T),union(S,U)) ==
  union(add_element(X,S),union(T,U))
intersection(intersection(S,T),union(S,U)) ==
  intersection(S,T)
intersection(intersection(S,T),union(T,U)) ==
  intersection(S,T)
intersection(intersection(S,U),intersection(T,V)) ==
  intersection(intersection(S,T),intersection(U,V))
intersection(intersection(S,U),union(T,U)) ==
  intersection(S,U)
intersection(intersection(T,U),union(S,T)) ==
  intersection(T,U)
```

Terrible! Notice how many of the equations are just simple variations of each other. By contrast, the current implementation of QuickSpec, using the pruning algorithm described in section 2.5, returns these 17 laws only:

```
1.  intersection(T,S) == intersection(S,T)
2.  union(T,S) == union(S,T)
3.  intersection(S,S) == S
4.  intersection(S,new()) == new()
5.  union(S,S) == S
6.  union(S,new()) == S
7.  add_element(Y,add_element(X,S)) ==
      add_element(X,add_element(Y,S))
8.  intersection(T,intersection(S,U)) ==
      intersection(S,intersection(T,U))
9.  union(S,add_element(X,T)) == add_element(X,union(S,T))
10. union(T,union(S,U)) == union(S,union(T,U))
11. intersection(S,add_element(X,S)) == S
12. intersection(S,union(S,T)) == S
13. union(S,intersection(S,T)) == S
```

```
14. intersection(add_element(X,S),add_element(X,T)) ==
      add_element(X,intersection(S,T))
15. intersection(union(S,T),union(S,U)) ==
      union(S,intersection(T,U))
16. union(add_element(X,S),add_element(X,T)) ==
      add_element(X,union(S,T))
17. union(intersection(S,T),intersection(S,U)) ==
      intersection(S,union(T,U))
```

This is still not ideal—there is some repetition between `add_element` and `union` (more on that in section 2.8), but is *much* better.

Comparison with the current algorithm The rewriting-based pruning algorithm, although often reasonable, is brittle and unpredictable. It depends on ordering the terms in *exactly the right way* so that all the laws we want to erase can be proved purely by simplification—we can never rewrite a smaller term to a larger term. It is far too sensitive to the exact term order we use (to get good results we had to use a carefully-tuned, delicate heuristic), and blows up when there is no natural term order to use, such as when we have commutativity or distributivity laws.

Our congruence closure-based algorithm, by contrast, is supremely well-behaved. It has no problem with commutativity laws, since it doesn't need to orient equations. It is quite insensitive to the equation ordering, unlike the rewriting-based algorithm. It works very well on most examples; more importantly, we know of no examples where it does *badly*: we can rely on it to produce a reasonable set of laws whatever program we give QuickSpec.

2.8 Definitions

Recall our sets example from section 2.7. We have four operators, `new` that returns the empty set, `add_element` that adds an element to a set, and `union` and `intersection`. Running QuickSpec on this signature, we get slightly unsatisfactory results:

```
1.  intersection(T,S) == intersection(S,T)
2.  union(T,S) == union(S,T)
3.  intersection(S,S) == S
4.  intersection(S,new()) == new()
5.  union(S,S) == S
6.  union(S,new()) == S
7.  add_element(Y,add_element(X,S)) ==
```

```
      add_element(X,add_element(Y,S))
8.  intersection(T,intersection(S,U)) ==
      intersection(S,intersection(T,U))
9.  union(S,add_element(X,T)) == add_element(X,union(S,T))
10. union(T,union(S,U)) == union(S,union(T,U))
11. intersection(S,add_element(X,S)) == S
12. intersection(S,union(S,T)) == S
13. union(S,intersection(S,T)) == S
14. intersection(add_element(X,S),add_element(X,T)) ==
      add_element(X,intersection(S,T))
15. intersection(union(S,T),union(S,U)) ==
      union(S,intersection(T,U))
16. union(add_element(X,S),add_element(X,T)) ==
      add_element(X,union(S,T))
17. union(intersection(S,T),intersection(S,U)) ==
      intersection(S,union(T,U))
```

These results state everything we would like to know about union and intersection, but there are perhaps more laws than we would like to see. Several laws appear in two variants, one for `union` and one for `add_element`.

This suggests that `union` and `add_element` are similar somehow. And indeed they are: `add_element` is the special case of `union` where one set is a singleton set ($S \cup \{x\}$). The way to reduce the number of equations is to replace `add_element` by a function `unit` that returns a singleton set. This function is simpler, so we expect better laws, but the API remains as expressive as before because `add_element` can be defined using `unit` and `union`. If we do that we get fewer laws:

```
1.  intersection(T,S) == intersection(S,T)
2.  union(T,S) == union(S,T)
3.  intersection(S,S) == S
4.  intersection(S,new()) == new()
5.  union(S,S) == S
6.  union(S,new()) == S
7.  intersection(T,intersection(S,U)) ==
      intersection(S,intersection(T,U))
8.  union(T,union(S,U)) == union(S,union(T,U))
9.  intersection(S,union(S,T)) == S
10. union(S,intersection(S,T)) == S
11. intersection(union(S,T),union(S,U)) ==
```

```
      union(S,intersection(T,U))
12. union(intersection(S,T),intersection(S,U)) ==
      intersection(S,union(T,U))
```

Now all the laws are recognisable as standard set theory ones, so we should conclude that there is not much redundancy here. Much better!

This technique is more widely applicable: whenever we have a redundant operator we will often get better results from QuickSpec if we remove it from the signature. QuickSpec in fact alerts us that an operator is redundant by printing out a *definition* of that operator in terms of other operators. In our case, when we ran QuickSpec the first time it also printed the following:

```
add_element(X,S) := union(S,add_element(X,new()))
```

In other words, `add_element(X,S)` is the union of `S` and the singleton set `{X}`, which we can construct with `add_element(X,new())`.

What QuickSpec looks for when it searches for definitions is a pair of equal terms in the equivalence relation satisfying the following conditions:

- One term must be a function call with all arguments distinct variables. In our case, this is `add_element(X,S)`. This is the left-hand side of the definition.
- The definition should not be circular; for example, we should not emit `union(S,T) := union(T,S)` as a definition. One possibility would be to forbid the right-hand side of a definition from referring to the function we're trying to define. However, this is too restrictive: in the definition of `add_element`, we use `add_element` on the right-hand side but we use a *special case* of `add_element` to construct a singleton set. We capture this by allowing the right-hand side of the definition to call the function it defines, but with one restriction: there must be a variable on the left-hand side of the definition that does not appear in the "recursive" call. In our case, the left-hand side mentions the variable `S` and the recursive call to `add_element` does not, so we conclude that the recursive call is a special case of `add_element` rather than a circular definition.

2.9 The depth optimisation

QuickSpec includes one optimisation to reduce the number of terms generated. We will first motivate the optimisation and then explain it in more detail.

Suppose we have run QuickSpec on an API of boolean operators with a depth limit of 2, giving (among others) the law `x&&x==x`. But now, suppose we want to increase the depth limit on terms from 2 to 3. Using the algorithm described above, we would first generate all terms of depth 3, including such ones as `x&&y` and `(x&&x)&&y`. But these two terms are obviously equivalent (since we know that `x&&x==x`), we won't get any more laws by generating both of them, and we ought to generate only `x&&y` and not `(x&&x)&&y`.

The observation we make is that, if two terms are equal (like `x&&x` and `x` above), we ought to pick one of them as the "canonical form" of that expression; we avoid generating any term that has a non-canonical form as a subterm. In this example, we don't generate `(x&&x)&&y`, because it has `x&&x` as a subterm.

The depth optimisation applies this observation, and works as follows. If we want to generate all terms up to depth 3, say, we first generate all terms up to depth 2 and sort them into equivalence classes by testing. The representatives of those classes we intend to be the "canonical forms" we mentioned above. Then, for terms of depth 3, we generate only those terms for which all the direct subterms are the representatives of their equivalence class. In the example above, we have an equivalence class `{x, x&&x}`; `x` is the representative. So we will generate terms that contain `x` as a direct subterm but not ones that contain `x&&x`.

We can justify why this optimisation is sound. If we choose not to generate a term `t` with canonical form `t'`, and if testing would have revealed an equation `t==u`, we will also generate an equation `t'==u`.[5] We also will have generated laws that imply `t==t'`,and therefore `t==u` is redundant. (In fact, the pruner would've filtered out `t==u`.)

This optimisation makes a very noticeable difference to the number of terms generated. For a large list signature, the number of terms goes down from 21266 to 7079. For booleans there is a much bigger difference, since so many terms are equal: without the depth optimisation we generate 7395 terms, and with it 449 terms. Time-wise, the method becomes an order of magnitude faster.

[5] This relies on `t'` not having greater depth than `t`, which requires the term ordering to always pick the representative of an equivalence class as a term with the smallest depth.

2.10 Generating Test Data

As always with random testing tools, the quality of the test data determines the quality of the generated equations. As such, it is important to provide good test data generators, that fit the program at hand. In our property-based random testing tool QuickCheck [Claessen and Hughes, 2000], we have a range of test data generators for standard types, and a library of functions for building custom generators. QuickSpec simply reuses QuickCheck's random data generators.

As an example of what can happen if we use an inappropriate generator, consider generating laws for an API including the following functions:

```
isPrefixOf :: [Elem] -> [Elem] -> Bool
null       :: [Elem] -> Bool
```

If one is not careful in defining the list generator, QuickSpec might end up producing the law `isPrefixOf xs ys == null xs`. Why? Because for two randomly generated lists, it is very unlikely that one is a prefix of the other, unless the first is empty. So, there is a risk that the interesting test cases that separate `isPrefixOf xs ys` and `null xs` will not be generated. The problem can be solved by making sure that the generator used for random lists is likely to pick the list elements from a small domain. Thus, just as in using QuickCheck, creating custom generators is sometimes necessary to get correct results.

We have to point out that this does not happen often even if we are careless about test data generation: as noted earlier, in our implementation we keep on refining the equivalence relation until it has been stable for 200 iterations, which gives QuickSpec a better chance of falsifying hard-to-falsify equations.

3 Case Studies

In this section, we present two case studies using QuickSpec. Our goal is primarily to derive *understanding* of the code we test. In many cases, the specifications generated by QuickSpec are initially disappointing—but by *extending the signature with new operations* we are able to arrive at concise and perspicuous specifications. Arguably, selecting the right operations to specify is a key step in formulating a good specification, and one way to see QuickSpec is as a tool to support exploration of this design space.

3.1 Case Study #1: Leftist Heaps in Haskell

A *leftist heap* [Okasaki, 1998] is a data structure that implements a priority queue. A leftist heap provides the usual heap operations:

```
empty :: Heap
isEmpty :: Heap -> Bool
insert :: Elem -> Heap -> Heap
findMin :: Heap -> Elem
deleteMin :: Heap -> Heap
```

When we tested this signature with the variables `h, h1, h2 :: Heap` and `x, y, z :: Elem` then QuickSpec generated a rather incomplete specification. The specification describes the behaviour of `findMin` and `deleteMin` on empty and singleton heaps:

```
findMin empty == undefined
findMin (insert x empty) == x
deleteMin empty == undefined
deleteMin (insert x empty) == empty
```

It shows that the order of insertion into a heap is irrelevant:

```
insert y (insert x h) == insert x (insert y h),
```

Apart from that, it only contains the following equation:

```
isEmpty (insert x h1) == isEmpty (insert x h)
```

This last equation is quite revealing—obviously, we would expect both sides to be `False`, which explains why they are equal. But why doesn't QuickSpec just print the equation `isEmpty (insert x h) == False`? The reason is that `False` is not in our signature! When we add it to the signature, then we do indeed obtain the simpler form instead of the original equation above.[6]

Generalising a bit, since `isEmpty` returns a `Bool`, it's certainly sensible to give QuickSpec operations that manipulate booleans. We added the remaining boolean connectives `True`, `&&`, `||` and `not`; one new law appeared that we couldn't express before, `isEmpty empty == True`.

[6] For completeness, we will list all of the new laws that QuickSpec produces every time we change the signature.

Merge

Leftist heaps actually provide one more operation than those we encountered so far: merging two heaps.

```
merge :: Heap -> Heap -> Heap
```

If we run QuickSpec on the new signature, we get the fact that `merge` is commutative and associative and has `empty` as a unit element:

```
merge h1 h == merge h h1
merge h1 (merge h h2) == merge h (merge h1 h2)
merge h empty == h
```

We get nice laws about `merge`'s relationship with the other operators:

```
merge h (insert x h1) == insert x (merge h h1)
isEmpty h && isEmpty h1 == isEmpty (merge h h1)
```

We also get some curious laws about merging a heap with itself:

```
findMin (merge h h) == findMin h
merge h (deleteMin h) == deleteMin (merge h h)
```

These are *all* the equations that are printed. Note that there are no redundant laws here. As mentioned earlier, our testing method guarantees that this set of laws is *complete*, in the sense that any valid equation over our signature, which is not excluded by the depth limit, follows from these laws.

With Lists

We can get useful laws about heaps by relating them to a more common data structure, *lists*. First, we need to extend the signature with operations that convert between heaps and lists:

```
fromList :: [Elem] -> Heap
toList :: Heap -> [Elem]
```

`fromList` turns a list into a heap by folding over it with the `insert` function; `toList` does the reverse, deconstructing a heap using `findMin` and `deleteMin`. We should also add a few list operations mentioned earlier:

```
(++) :: [Elem] -> [Elem] -> [Elem]
tail :: [Elem] -> [Elem]
(:) :: Elem -> [Elem] -> [Elem]
[] :: [Elem]
sort :: [Elem] -> [Elem]
```

and variables `xs, ys, zs :: [Elem]`. Now, QuickSpec discovers many new laws. The most striking one is

```
toList (fromList xs) == sort xs.
```

This is the definition of heapsort! The other laws indicate that our definitions of `toList` and `fromList` are sensible:

```
sort (toList h) == toList h
fromList (toList h) == h
fromList (sort xs) == fromList xs
fromList (ys++xs) == fromList (xs++ys)
```

The first law says that `toList` produces a sorted list, and the second one says that `fromList . toList` is the identity (up to `==` on heaps, which actually applies `toList` to each operand and compares them!). The other two laws suggest that the order of `fromList`'s input doesn't matter.

We get a definition by pattern-matching of `fromList` (read the second and third equations from right to left):

```
fromList [] == empty
insert x (fromList xs) == fromList (x:xs)
merge (fromList xs) (fromList ys) == fromList (xs++ys)
```

We also get a family of laws relating heap operations to list operations:

```
toList empty == []
head (toList h) == findMin h
toList (deleteMin h) == tail (toList h)
```

We can think of `toList h` as an abstract model of `h`—all we need to know about a heap is the sorted list of elements, in order to predict the result of any operation on that heap. The heap itself is just a clever representation of that sorted list of elements.

The three laws above define `empty`, `findMin` and `deleteMin` by how they act on the sorted list of elements—the model of the heap. For

example, the third law says that applying `deleteMin` to a heap corresponds to taking the `tail` in the abstract model (a sorted list). Since `tail` is obviously the correct way to remove the minimum element from a sorted list, this equation says exactly that `deleteMin` is correct![7]

So these three equations are a *complete* specification of the three functions `empty`, `findMin` and `deleteMin`!

If we want to extend this to a complete specification of heaps, we must add operators to insert an element into a sorted list, to merge two sorted lists, and to test if a sorted list is empty…

```
insertL :: Elem -> [Elem] -> [Elem]
mergeL :: [Elem] -> [Elem] -> [Elem]
null :: [Elem] -> Bool
```

…and our reward is three laws asserting that the functions `insert`, `merge` and `isEmpty` are correct:

```
toList (insert x h) == insertL x (toList h)
mergeL (toList h) (toList h1) == toList (merge h h1)
null (toList h) == isEmpty h
```

We also get another law about `fromList` to go with our earlier collection. This one says essentially that `mergeL xs ys` contains each of the members of both `xs` and `ys` exactly once:

```
fromList (mergeL xs ys) == fromList (xs++ys)
```

This section highlights the importance of choosing a rich set of operators when using QuickSpec. There are often useful laws about a library that mention functions from unrelated libraries; the more such functions we include, the more laws QuickSpec can find. In the end, we got a complete specification of heaps (and heapsort, as a bonus!) by including list functions in our testing.

It's not always obvious *which* functions to add to get better laws. In this case, there were several reasons for choosing lists: they're well-understood, there are operators that convert heaps to and from lists, and sorted lists form a model of priority queues.

[7] This style of specification is not new and goes back to Hoare [1972].

Buggy Code

What happens when the code under test has a bug? To find out, we introduced a fault into `toList`. The buggy version of `toList` doesn't produce a sorted list, but rather the elements of the heap in an arbitrary order.

We were hoping that some laws would fail, and that QuickSpec would produce *specific instances* of some of those laws instead. This happened: whereas before, we had many useful laws about `toList`, afterwards, we had only two:

```
toList empty == []
toList (insert x empty) == x:[]
```

Two things stand out here: first, we know that the buggy `toList` does not produce a sorted result, otherwise we would get the law `sort (toList h) == toList h`. Second, we *only get equations about empty and singleton heaps*, not about heaps of arbitrary size. QuickSpec is unable to find *any* specification of `toList` on non-trivial heaps, which suggests that the buggy `toList` *has* no simple specification.

A trick

We finish with a "party trick": getting QuickSpec to discover how to implement `insert` and `deleteMin`. We hope to run QuickSpec and see it print equations of the form `insert x h = ?` and `deleteMin h = ?`.

We need to prepare the trick first; if we just run QuickSpec straight away, we won't get either equation. There are two reasons, each of which explains the disappearance of one equation.

First, it's impossible to implement `deleteMin` using only the leftist heap API, so there's no equation for QuickSpec to print. To give QuickSpec a chance, we need to reveal the representation of leftist heaps; they're really binary trees. So we add the functions

```
leftBranch :: Heap -> Heap
rightBranch :: Heap -> Heap
```

to the signature. Of course, no implementation of leftist heaps would export these functions, this is only for the trick.

Secondly, QuickSpec won't bother to print out the definition of `insert`: it's easily derivable from the other laws, so QuickSpec considers it boring. Actually, in most ways, it *is* pretty boring; the one thing that

makes it interesting is that it defines `insert`, but QuickSpec takes no notice of that.

Fortunately, we have a card up our sleeve: QuickSpec prints a list of *definitions*, equations that define an operator in terms of other operators, as we saw in section 2.8. The real purpose of this is to suggest redundant operators, but we will use it to see the definition of `insert` instead.

Finally, we also need to be careful—previously, we were treating our heap as an *abstract data type*, so that two heaps would be equal if they had the same elements. But `leftBranch` and `rightBranch` peek into the internals of the heap, so they can distinguish heaps that are morally the same. So we had better tell QuickSpec that equality should check the representation of the heap and not its contents.

Everything in place at last, we run QuickSpec. And—hey presto!—out come the equations

```
insert x h = merge h (insert x empty)
deleteMin h = merge (leftBranch h) (rightBranch h)
```

That is, you can insert an element by merging with a unit heap that just contains that element, or delete the minimum element—which happens to be stored at the root of the tree—by merging the root's branches.

3.2 Case Study #2: Understanding a Fixed Point Arithmetic Library in Erlang

We used QuickSpec to try to understand a library for fixed point arithmetic, developed by a South African company, which we were previously unfamiliar with. The library exports 16 functions, which is rather overwhelming to analyze in one go, so we decided to generate equations for a number of different subsets of the API instead. In this section, we give a detailed account of our experiments and developing understanding.

Before we could begin to use QuickSpec, we needed a QuickCheck generator for fixed point data. We chose to use one of the library functions to ensure a valid result, choosing one which seemed able to return arbitrary fixed point values:

```
fp() -> ?LET({N,D},{largeint(),nat()},from_minor_int(N,D)).
```

That is, we call `from_minor_int` with random arguments. We suspected that `D` is the precision of the result—a suspicion that proved to be correct.

Addition and Subtraction

We began by testing the add operation, deriving commutativity and associativity laws as expected. Expecting laws involving zero, we defined

```
zero() -> from_int(0)
```

and added it to the signature, obtaining as our reward a unit law,

```
add(A,zero()) == A.
```

The next step was to add subtraction to the signature. However, this led to several very similar laws being generated—for example,

```
add(B,add(A,C)) == add(A,add(B,C))
add(B,sub(A,C)) == add(A,sub(B,C))
sub(A,sub(B,C)) == add(A,sub(C,B))
sub(sub(A,B),C) == sub(A,add(B,C))
```

To relieve the problem, we added another derived operator to the signature instead:

```
negate(A) -> sub(zero(),A).
```

and observed that the earlier family of similar laws was no longer generated, replaced by a single one, `add(A,negate(B)) == sub(A,B)`. Thus by *adding* a new auxiliary function to the signature, `negate`, we were able to *reduce* the complexity of the specification considerably.

After this new equation was generated by QuickSpec, we tested it extensively using *QuickCheck*. Once confident that it held, we could safely *replace* `sub` in our signature by `add` and `negate`, without losing any other equations. Once we did this, we obtained a more useful set of new equations:

```
add(negate(A),add(A,A)) == A
add(negate(A),negate(B)) == negate(add(A,B))
negate(negate(A)) == A
negate(zero()) == zero()
```

These are all very plausible—what is striking is the *absence* of the following equation:

```
add(A,negate(A)) == zero()
```

When an expected equation like this is missing, it's easy to formulate it as a QuickCheck property and find a counterexample, in this case `{fp,1,0,0}`. We discovered by experiment that `negate({fp,1,0,0})` is actually the same value! This strongly suggests that this is an alternative representation of zero (`zero()` evaluates to `{fp,0,0,0}` instead).

Zero is not equal to zero

It is reasonable that a fixed point arithmetic library should have different representations for zero of different precisions, but we had not anticipated this. Moreover, since we want to derive equations involving zero, the question arises of *which zero* we would like our equations to contain! Taking our cue from the missing equation, we introduced a new operator `zero_like(A) -> sub(A,A)` and then derived not only `add(A,negate(A)) == zero_like(A)` but a variety of other interesting laws. These two equations suggest that the result of `zero_like` depends only on the number of decimals in its argument,

```
zero_like(from_int(I)) == zero()
zero_like(from_minor_int(J,M)) ==
  zero_like(from_minor_int(I,M))
```

this equation suggests that the result has the *same* number of decimals as the argument,

```
zero_like(zero_like(A)) == zero_like(A)
```

while these two suggest that the number of decimals is preserved by arithmetic.

```
zero_like(add(A,A)) == zero_like(A)
zero_like(negate(A)) == zero_like(A)
```

It is not in general true that `add(A,zero_like(B)) == A` which is not so surprising—the precision of `B` affects the precision of the result. QuickSpec does find a more restricted property, `add(A,zero_like(A)) == A`.

The following equations suggest that the precision of the results of `add` and `negate` depend only on the precision of the arguments, not their values:

```
add(zero_like(A),zero_like(B)) == zero_like(add(A,B))
negate(zero_like(A)) == zero_like(A)
```

Multiplication and Division

When we added multiplication and division operators to the signature, then we followed a similar path, and were led to introduce `reciprocal` and `one_like` functions, for similar reasons to `negate` and `zero_like` above. One interesting equation we discovered was this one:

```
divide(one_like(A),reciprocal(A)) ==
  reciprocal(reciprocal(A))
```

The equation is clearly true, but why does it say `reciprocal(reciprocal(A))` instead of just `A`? The reason is that the left-hand side raises an exception if `A` is zero, and so the right-hand side must do so also—which `reciprocal(reciprocal(A))` does.

We obtain many equations that express things about the precision of results, such as

```
multiply(B,zero_like(A)) == zero_like(multiply(A,B))
multiply(from_minor_int(I,N),from_minor_int(J,M)) ==
           multiply(from_minor_int(I,M),from_minor_int(J,N))
```

where the former expresses the fact that the precision of the zero produced depends both on `A` and `B`, and the latter expresses

$$\mathrm{i} \times 10^{-\mathrm{m}} \times \mathrm{j} \times 10^{-\mathrm{n}} = \mathrm{i} \times 10^{-\mathrm{n}} \times \mathrm{j} \times 10^{-\mathrm{m}}$$

That is, it is in a sense the commutativity of multiplication in disguise.

One equation we expected, but did *not* see, was the distributivity of multiplication over addition. Alerted by its absence, we formulated a corresponding QuickCheck property,

```
prop_multiply_distributes_over_add() ->
    ?FORALL({A,B,C},{fp(),fp(),fp()},
        multiply(A,add(B,C)) ==
          add(multiply(A,B),multiply(A,C))).
```

and used it to find a counterexample:

```
A = {fp,1,0,4}, B = {fp,1,0,2}, C = {fp,1,1,4}
```

We used the library's `format` function to convert these to strings, and found thus that $A = 0.4, B = 0.2, C = 1.4$. Working through the example, we found that multiplying `A` and `B` returns a representation of 0.1, and so we were alerted to the fact that `multiply` rounds its result to the precision of its arguments.

Understanding Precision

At this point, we decided that we needed to understand how the precision of results was determined, so we defined a function `precision` to extract the first component of an `{fp,...}` structure, where we suspected the precision was stored. We introduced a `max` function on naturals, guessing that it might be relevant, and (after observing the term `precision(zero())` in generated equations) the constant natural zero. QuickSpec then generated equations that tell us rather precisely how the precision is determined, including the following:

```
max(precision(A),precision(B)) == precision(add(A,B))
precision(divide(zero(),A)) == precision(one_like(A))
precision(from_int(I)) == 0
precision(from_minor_int(I,M)) == M
precision(multiply(A,B)) == precision(add(A,B))
precision(reciprocal(A)) == precision(one_like(A))
```

The first equation tells us the addition uses the precision of whichever argument has the most precision, and the fifth equation tells us that multiplication does the same. The second and third equations confirm that we have understood the representation of precision correctly. The second and sixth equations reveal that our definition of `one_like(A)` raises an exception when `A` is zero—this is why we do not see

```
precision(one_like(A)) == precision(A).
```

The second equation is more specific than we might expect, and in fact it is true that

```
precision(divide(A,B)) ==
  max(precision(A),precision(one_like(B)))
```

but the right-hand side exceeds our depth limit, so QuickSpec cannot discover it.

If we could discover conditional equations, then QuickSpec might discover instead that

```
B/=zero_like(B) ==>
  precision(divide(A,B)) == precision(add(A,B))
```

a property which we verified with QuickCheck.

3.3 Adjusting Precision

The library contained two operations whose meaning we could not really guess from their names, `adjust` and `shr`. Adding `adjust` to the signature generated a set of equations including the following:

```
adjust(A,precision(A)) == A
precision(adjust(A,M)) == M
zero_like(adjust(A,M)) == adjust(zero(),M)
adjust(zero_like(A),M) == adjust(zero(),M)
```

These equations make it fairly clear that `adjust` sets the precision of its argument. We also generated an equation relating double to single adjustment:

```
adjust(adjust(A,M),0) == adjust(A,0)
```

We generalised this to

```
N =< M ==> adjust(adjust(A,M),N) == adjust(A,N),
```

a law which QuickSpec might well have generated if it could produce conditional equations. We tested the new equation with QuickCheck, and discovered it to be false! The counterexample QuickCheck found shows that the problem is caused by rounding: adjusting 0.1045 to three decimal places yields 0.105, and adjusting this to two decimals produces 0.11. Adjusting the original number to two decimals in one step produces 0.10, however, which is different. In fact, the original equation that QuickSpec found above is also false—but several hundred tests are usually required to find a counterexample. This shows the importance of testing the most interesting equations that QuickSpec finds more extensively—occasionally, it does report falsehoods. Had we written a test data generator that tried to provoke interesting rounding behaviours we mightn't have encountered these false equations.

3.4 `shr`: Problems with Partiality

Adding `shr` to the signature, too, we at first obtained several rather complex equations, of which this is a typical example:

```
adjust(shr(zero(),I),precision(A)) == shr(zero_like(A),I)
```

All the equations had one thing in common: `shr` appeared on both sides of the equation, with the same second argument. Eventually we realised

why: `shr` is a *partial* function, which raises an exception when its second argument is negative—so QuickSpec produced equations with `shr` on both sides so that exceptions would be raised in the same cases. We changed the signature to declare `shr`'s second argument to be `nat()` rather than `int()`, whereupon QuickSpec produced simple equations as usual.

QuickSpec told us how the precision of the result is determined:

```
precision(shr(A,M)) == precision(A)
```

Other informative equations were

```
shr(shr(A,N),M) == shr(shr(A,M),N)
shr(A,0) == A
shr(zero_like(A),M) == zero_like(A)
```

and, after we introduced addition on naturals,

```
shr(shr(A,M),N) == shr(A,M+N)
```

We began to suspect that `shr` implemented a right shift, and to test this hypothesis we formulated the property

```
prop_shr_value() ->
  ?FORALL({N,A},{nat(),fp()},
          shr(multiply(A,pow(10,N)),N) == A).
```

and after 100,000 successful tests concluded that our hypothesis was correct.

Summing Up

Overall, we found QuickSpec to be a very useful aid in developing an understanding of the fixed point library. Of course, we could simply have formulated the expected equations as QuickCheck properties, and tested them without the aid of QuickSpec. However, this would have taken very much longer, and because the work is fairly tedious, there is a risk that we might have forgotten to include some important properties. QuickSpec automates the tedious part, and allowed us to spot missing equations quickly.

Of course, QuickSpec also generates *unexpected* equations, and these would be much harder to find using QuickCheck. In particular, when investigating functions such as `adjust`, where we initially had

little idea of what they were intended to do, then it would have been very difficult to formulate candidate QuickCheck properties in advance.

Although QuickSpec can run given any signature, we discovered that if QuickSpec is used without sufficient thought, the result is often disappointing. We needed to use our ingenuity to extend the signature with useful auxiliaries, such as `negate`, `precision`, `+` and `max`, to get the best out of QuickSpec.

Since QuickSpec is unsound, it may generate equations which are not true, as it did once in our case study. However, even these false equations can be quite informative, since they are *nearly* true—they are simple statements which passed a few hundred test cases. They are thus likely *misconceptions* about the code, and formulating them, then discovering their falsehood by more extensive testing, contributes in itself to understanding of the code. ("You might think that such-and-such holds, but oh no, consider this case!"). We regularly include such *negative properties* in QuickCheck specifications, to prevent the same misconception arising again. QuickSpec runs relatively few tests of each equation (several hundred), and so, once the most interesting equations have been selected, then it is valuable to QuickCheck them many more times to make sure that they are true. It can also be worthwhile, just as in random testing in general, to tweak the test data generator to get a better distribution of random data.

QuickSpec's equation filtering mostly did a fine job of reducing the number of generated equations. However, it sometimes helped to alter the signature to make QuickSpec's job easier—such as replacing `sub` by the simpler function `negate`.

The case study highlights the difficulties that partial functions can cause: our requirement that the left and right-hand sides of an equation must raise exceptions in *exactly* the same cases leads QuickSpec to generate impenetrable equations containing complex terms whose only purpose is to raise an exception in the right cases. QuickSpec cannot currently find equations that hold, provided preconditions are fulfilled. It would be useful to report "weak equations" too, whose left and right-hand sides are equal whenever both are defined. However, it is not clear how QuickSpec should prune such equations, since "weak equality" is not an equivalence relation and the reasoning principles for "weak equations" are not obvious. At the very least, QuickSpec should inform us when it discovers that a function is partial.

Another way to address partiality would be to generate conditional equations with the function preconditions as the condition. Indeed, this would be a generally useful extension, and the case study also highlights

other examples where conditional equations would be useful.

4 Related Work

As mentioned earlier, the existing work that is most similar to ours is Henkel et al. [2007]; a tool for discovering algebraic specifications from Java classes. They generate terms and evaluate them, dynamically identify terms which are equal, then generate equations and filter away redundant ones. There are differences in the kind of equations that can be generated, which have been discussed earlier.

The most important difference in the two approaches is the fact that they start by generating a large set of *ground* terms when searching for equations, which they then test, filter, generalize and prune. So, their initial set both represents the possible terms that can occur in the equations, and the "test cases" that are run. The term set they use thus becomes extremely large, and in order to control its size, they use heuristics such as only generating random subsets of all possible terms, and restricting values to very small domains. This choice not only sacrifices completeness, but also predictability and controllability. In contrast, we always generate *all* possible terms that can occur in equations (keeping completeness), and then use random testing to gather knowledge about these terms. If we end up with too few equations, we can increase the number of terms; if we end up too many equations, we can increase the number of random tests.

There are other differences as well. They test terms for operational equivalence, which is quite expensive; we use fast structural equivalence or a user-specified equality test. They use a heuristic term-rewriting method for pruning equations which will not handle structural properties well (we note that their case studies do not include commutative and associative operators, which we initially found to be extremely problematic); we use a predictable congruence closure algorithm. We are able to generate equations relating higher-order functions; working in Java, this was presumably not possible. They observe—as we do—that conditional equations would be useful, but neither tool generates them. Our tool appears to be faster (our examples take seconds to run, while comparable examples in their setting take hours). It is unfortunately rather difficult to make a fair comparison between the efficacy and performance of the two approaches, because their tool and examples are not available for download.

Daikon [Ernst et al., 2007] is a tool for inferring likely invariants

in C, C++, Java or Perl programs. Daikon observes program variables at selected program points during testing, and applies machine learning techniques to discover relationships between them. For example, Daikon can discover linear relationships between integer variables, such as array indices. Agitar's commercial tool based on Daikon generates test cases for the code under analysis automatically [Boshernitsan et al., 2006]. However, Daikon will not discover, for example, that `reverse(reverse(Xs)) == Xs`, unless such a double application of `reverse` appears in the program under analysis. Whereas Daikon discovers invariants that hold at existing program points, QuickSpec discovers equations between arbitrary terms constructed using an API. This is analogous to the difference between *assertions* placed in program code, and the kind of *properties* which QuickCheck tests, that also invoke the API under test in interesting ways. While Daikon's approach is ideal for imperative code, especially code which loops over arrays, QuickSpec is perhaps more appropriate for analysing pure functions.

Inductive logic programming (ILP) [Muggleton and de Raedt, 1994] aims to infer logic programs from examples—specific instances—of their behaviour. The user provides both a collection of true statements and a collection of false statements, and the ILP tool finds a program consistent with those statements. Our approach only uses *false* statements as input (inequality is established by testing), and is optimized for deriving equalities.

In the area of *Automated Theorem Discovery* (ATD), the aim is to emulate the human theorem discovery process. The idea can be applied to many different fields, such as mathematics, physics, but also formal verification. An example of an ATD system for mathematicians is MATHsAiD [McCasland and Bundy, 2006]. The system starts by generating a finite set of *hypotheses*, according to some syntactical rules that capture typical mathematical thinking, for example: if we know $A \Rightarrow B$, we should also check if $B \Rightarrow A$, and if not, under what conditions this holds. Theorem proving techniques are used to select theorems and patch non-theorems. Since this leads to many theorems, a filtering phase decides if theorems are interesting or not, according to a number of different predefined "tests". One such test is the simplicity test, which compares theorems for simplicity based on their proofs, and only keeps the simplest theorems. The aim of their filtering is quite different from ours (they want to filter out theorems that mathematicians would have considered trivial), but the motivation is the same; there are too many theorems to consider.

QuickCheck is our own tool for random testing of functional pro-

grams, originally for Haskell [Claessen and Hughes, 2000] and now in a commercial version for Erlang [Arts et al., 2006]. QuickCheck tests *properties* such as the equations that QuickSpec discovers, so one application for QuickSpec is to quickly generate a QuickCheck test suite. However, QuickCheck supports a more general property language, including conditional properties and specifications for functions with side-effects [Claessen and Hughes, 2002, Hughes, 2007]. Both implementations of QuickSpec use QuickCheck to generate random test data; this allows users to exert fine control over the selection of test data by specifying an appropriate QuickCheck generator.

5 Conclusions and Future Work

We have presented a new tool, QuickSpec, which can automatically generate algebraic specifications for functional programs. Although simple, it is remarkably powerful. It can be used to aid program understanding, or to generate a QuickCheck test suite to detect changes in specification as the code under test evolves. We are hopeful that it will enable more users to overcome the barrier that formulating properties can present, and discover the benefits of QuickCheck-style specification and testing.

For future work, we plan to generate conditional equations. In some sense, these can be encoded in what we already have by specifying new custom types with appropriate operators. For example, if we want `x<=y` to occur as a precondition, we might introduce a type `AscPair` of "pairs with ascending elements", and add the functions

```
smaller, larger :: AscPair -> Int
```

and the variable `p :: AscPair` to the API. A conditional equation we could then generate is:

```
isSorted (smaller p : larger p : xs) ==
  isSorted (larger p : xs)
```

(Instead of the perhaps more readable `x<=y ==> isSorted (x:y:xs) == isSorted (y:xs)`.) But we are still investigating the limitations and applicability of this approach.

Another class of equations we are looking at is equations between program fragments that can have side effects. Our idea is to represent a program fragment by a monadic expression, or similar, and use QuickSpec's existing functionality to derive laws for these fragments. We have a prototype implementation of this but more work is needed.

The Erlang version of QuickSpec uses structural equality in the generated equations, which means that terms that may evaluate to different representations of the same abstract value are considered to be different, for example causing some of the unexpected results in section 3.2. The Haskell version uses the `(==)` operator, defined in the appropriate `Eq` instance. However, this is unsafe unless `(==)` is a congruence relation with respect to the operations in the API under test! QuickSpec has recently been extended to *test* for these properties while classifying terms, although we do not discuss this here.

It can be puzzling when an equation we expect to see is *missing*. A small extension would be to allow us to *ask* QuickSpec why an equation wasn't printed, and get either a proof of the equation (if it was pruned away) or a counterexample (if it was false). Both of these can quite easily be extracted from QuickSpec's data structures.

2

HipSpec: Automating Inductive Proofs using Theory Exploration

This paper will be published at CADE 2013 in Lake Placid.

Chapter 2

HipSpec: Automating Inductive Proofs using Theory Exploration

Koen Claessen Moa Johansson Dan Rosén
Nicholas Smallbone

Abstract

HipSpec is a system for automatically deriving and proving properties about functional programs. It uses a novel approach, combining theory exploration, counterexample testing and inductive theorem proving. HipSpec automatically generates a set of equational theorems about the available recursive functions of a program. These equational properties make up an algebraic specification for the program and can in addition be used as a background theory for proving additional user-stated properties. Experimental results are encouraging: HipSpec compares favourably to other inductive theorem provers and theory exploration systems.

1 Introduction

We are studying the problem of automatically proving algebraic properties of programs. Our aim is to build a tool that programmers can use to support software development. This paper describes current progress towards this goal, in particular addressing the problem of automating inductive proofs.

We work in a subset of the strongly typed functional programming language Haskell. Our subset consists of monomorphic, terminating programs without type classes or primitive types (like `Int`). The only data types are algebraic data types, functions and uninterpreted types. Removing these restrictions is ongoing work.

There are two key advantages of using Haskell as the input language. Firstly, a pure functional programming language is semantically simpler and thus easier to reason about than languages with side effects. Secondly, many Haskell programmers already use QuickCheck [Claessen and Hughes, 2000], a tool for property-based random testing, which means that many Haskell program are already annotated with formal properties (tested, but not proved).

The main obstacles one encounters when doing automated verification of functional programs are (1) when and how to apply induction, and (2) how to discover auxiliary lemmas or generalisations which may be required in inductive proofs. Let us look at a simple example. Consider the following Haskell program implementing the list reverse function in two different ways, `rev` and `qrev`. The latter uses a helper function `revacc` with an accumulating parameter which leads to a function with better time complexity. Their definitions are:

```
rev []     = []
rev (x:xs) = rev xs ++ [x]

revacc []     acc = acc
revacc (x:xs) acc = revacc xs (x:acc)

qrev xs = revacc xs []
```

A natural property one would like to verify is that the functions above produce the same result: $\forall$ `xs. rev xs = qrev xs`. Suppose we attempt to prove this by structural induction on `xs`. This will fail as the inductive hypothesis `rev as = qrev as` is too weak to prove `rev (a:as) = qrev (a:as)`. What is needed here is an additional lemma such as `rev xs++ys = revacc xs ys`, from which the original conjecture follows as a special case when `ys` happens to be the empty list. This is a typical example of the kind of generalisations which are required in proofs about functions with accumulator variables. One of the main challenges for inductive theorem provers is how to discover such lemmas automatically.

Current inductive theorem provers such as IsaPlanner [Dixon and Fleuriot, 2004], Zeno [Sonnex et al., 2011] and ACL2 [Kaufmann

et al., 2000] support a simple lemma discovery technique called *lemma calculation*, by which a new lemma is suggested by replacing some common subterm in a stuck goal by a variable. Although this technique works very well for many proofs, it is not enough for the above example, which cannot be automatically proved by these systems. The now defunct CLAM proof-planner had in addition a so-called *proof-critic* for discovering more complex generalisations [Ireland and Bundy, 1995], such as the one required in the example, but only if other basic lemmas were given by the user.

Our approach differs from the *top-down* manner in which the above systems work. Instead of waiting for the proof to somehow get stuck, we use *bottom-up* lemma discovery, or *theory exploration*. Our tool, called HipSpec, gets its name from its two subsystems which we developed previously: the automated inductive prover Hip [Rosén, 2012], and the conjecture generation system QuickSpec. Hip tries to prove a conjecture by enumerating all possible ways of doing structural induction over the free variables, and then calling an automated first-order prover to prove them. QuickSpec creates thousands of terms involving the functions of a given API, and computes equivalence classes over these terms by means of testing. Each pair of terms t_1, t_2 in an equivalence class gives rise to a conjecture $t_1 = t_2$.

HipSpec reads in a program, but besides trying to tackle any of the user-given properties, it asks QuickSpec to produce a list of conjectures about the program. HipSpec then sends these conjectures to Hip and those that are proved can be used as lemmas in subsequent proof-attempts. After this theory exploration phase, the properties stated by the programmer are tried, using all the proved lemmas as background theory.

There are several theory exploration systems which have been applied to discover theorems in inductive theories [Johansson et al., 2011, McCasland and Bundy, 2006, Montano-Rivas et al., 2012], but none have been fully integrated with an automated theorem prover in order to supply the prover with lemmas. Instead, these systems simply generate and prove a set of 'interesting' equations summarising the main properties about the program, which are then presented to the user. In fact, HipSpec may also be used in this manner without any user-stated properties.

Let us return to the example property about `rev`. HipSpec calls QuickSpec, which within a few seconds conjectures a set of equations about the functions involved. HipSpec feeds these to Hip, which tries to prove them. Those that can be proved without induction are redundant

and can be discarded; the lemmas needing induction are shown below[1]:

No	Conjecture	Proved using[2]
(1)	`xs++[] = xs`	`xs`
(2)	`(xs++ys)++zs = xs++(ys++zs)`	`xs`
(3)	`rev xs++rev ys = rev (ys++xs)`	`ys`, (1), (2)
(4)	`revacc (revacc xs ys) [] = revacc ys xs`	`xs`
(5)	`revacc (revacc xs ys) zs = revacc ys (xs++zs)`	`xs`
(6)	`revacc xs ys++zs = revacc xs (ys++zs)`	`zs`, (1), (2), (5)
(7)	`revacc xs (rev ys) = rev (ys++xs)`	`xs`, (1), (3), (6)

The original property is now easily proved: it follows directly from (7), letting `ys = []`, and the definition of `qrev`; induction is not even needed. Note that lemma (4) is not needed for proving the original property. Discovering some unnecessary lemmas is a (potentially disadvantageous) side-effect of the bottom-up approach.

Contributions. We augment the automated induction landscape with a new method which uses a bottom-up theory exploration approach to find auxiliary lemmas. This approach combines our own earlier work on conjecture generation based on testing (QuickSpec) and induction principle enumeration (Hip). By adding proof capabilities on top of QuickSpec we also get a system which can be used as a stand-alone theory exploration system.

Our hypothesis is that:

1. Algebraic equations constructed from terms up to a certain depth form a rich enough background theory for proving many algebraic properties about programs without specialised proof-critics.

2. A reasoning system for functional programs can be built on top of an automatic first-order theorem prover.

3. A system combining (1) and (2) can be used both as a theorem prover and as an efficient theory exploration system, producing background lemmas comparable to those appearing in human-created libraries.

The experimental results in this paper have so far confirmed this.

[1] The variables are implicitly universally quantified over total and finite values.
[2] This column shows the induction variables and which lemmas were used.

2 Implementation

Below we describe in more detail how Hip and QuickSpec work, and how they are combined in HipSpec.

2.1 Hip

Hip [Rosén, 2012] is an automatic tool for proving user-stated equality or implicational properties about Haskell programs. Hip starts by compiling the definitions in the program at hand to first-order logic. For each property stated in the program, it systematically applies different induction rules, yielding first-order proof obligations, which are tested for validity using off-the shelf automated first-order theorem provers. If one proof obligation succeeds, the original conjecture was valid. Thus, the first-order prover takes care of non-inductive reasoning, while Hip adds inductive reasoning at the meta-level. In the context of HipSpec, Hip is configured to apply structural induction up to a given depth on one or more variables. Hip, however, also supports co-inductive proof techniques such as fixed point induction. The focus of our work in HipSpec is currently not on proving termination, so we restrict ourselves by allowing only well-founded definitions, and put the responsibility on the end user to enforce this policy for now.

2.2 QuickSpec

QuickSpec (see Paper 1) conjectures equations about a functional program by means of testing. The user of QuickSpec provides a list of functions and their types, a random test data generator for each of the types involved, a set of variables (usually 2-3 per type), and a term depth limit (usually 3). QuickSpec starts by creating a set of terms, called the *universe*, consisting of all well-typed terms built from the functions and variables given, whose depth is within the given limit. It then partitions this universe into equivalence classes by running a finite number of random tests (usually 100); two terms will be in the same equivalence class if and only if they were equal for all tests. This equivalence relation in turn gives rise to a huge set of conjectured equations about the tested program (typically thousands or tens of thousands). For the sake of human users, QuickSpec also includes a final phase which prunes away equations that follow from simpler ones, leaving only a small core of equations from which all original equations follow. This core is usually presented to the user (usually 10-25 equations). However, when Hip-

Spec uses QuickSpec to generate lemmas, it does not use the pruning phase, because valuable lemmas may be pruned away. For example, even when an equation E_1 implies a more complex equation E_2, we can not necessarily discard E_2, because E_2 may be provable by induction whereas E_1 may not be. In fact, E_2 may very well be needed as a lemma to prove E_1! So, HipSpec considers the full set of equations produced by QuickSpec before pruning.

2.3 HipSpec

HipSpec's operation is illustrated in Figure 2.1. We start by running QuickSpec on the program source file, which generates a list of conjectures. We also translate the program source code to a first-order theory using Hip.

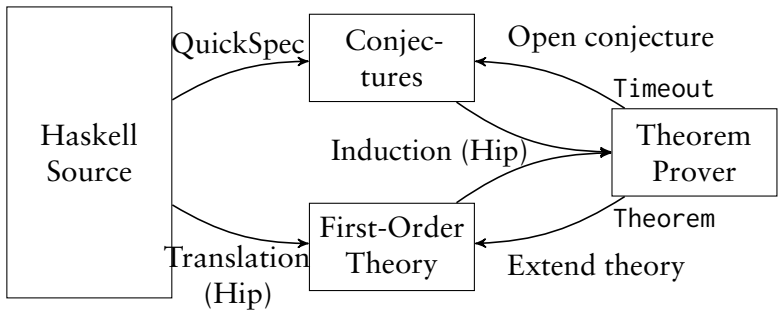


Figure 2.1: An overview of HipSpec.

HipSpec maintains three sets of equations: *active conjectures*, which we still need to consider, *failed conjectures*, which we have already tried to prove but failed, and *lemmas*, which we have managed to prove. The *first-order theory* in Figure 2.1 consists of Hip's translation of our program plus the current set of lemmas. Initially the active conjectures consist of all equations that QuickSpec found (even those that would have been removed by pruning), and the failed conjecture set and lemma set are empty.

The main loop works as follows:

1. Pick a conjecture c from the active conjecture set (using a heuristic described below).

2. Check if c follows from the lemmas found so far by equational reasoning only. If so, discard c, and re-iterate.

3. Otherwise, ask Hip to prove the conjecture by induction, using definitions and previously proved lemmas as background theory.

4. If Hip succeeds, move c to the lemma set, and move some failed conjectures back to the active conjectures (based on a heuristic described below).

5. If Hip does not succeed within a set timeout, move c to the failed conjectures.

The loop ends when the active conjecture set is empty.

Picking the conjecture The performance of HipSpec completely depends on one heuristic: which active conjecture to try to prove next. Our current heuristics are rather crude; more sophisticated techniques are further work.

Our basic strategy is to prove simpler equations before more complicated ones. We define simplicity as follows. A smaller term is simpler than a bigger term; if two terms have the same size, the term with more variables is simpler (because it might be more general). For example, `(x+y)+z=x+(y+z)` is simpler than `(x+x)+y=x+(x+y)`. The simplicity of an equation $t_1 = t_2$ is determined by whichever of t_1 and t_2 is the most complex.

We also take into account the call graph of the program. For example, if we are proving properties about the natural numbers, we prove as much as possible about + before starting on $*$, since $*$ calls +. More precisely, when choosing which conjecture to prove next, we pick the one whose call graph is the smallest; if two conjectures have the same size call graph, we pick the simplest one.

Discarding trivial consequences It is quite expensive to send every conjecture to Hip to be proved, when we may have thousands of them. Luckily, QuickSpec has a lightweight theorem prover based on congruence closure. This prover can efficiently answer questions of the form "given these lemmas, can I prove this equation?", replying either "yes" or "don't know".

Whenever we pick a conjecture, we check if this prover can prove it from the current lemmas without induction. If so, we just discard it. This filters out most trivial conjectures that are provable without induction.

Re-activating failed conjectures When we prove a lemma, we sometimes move some failed conjectures back to the active conjectures. HipSpec's rule is to wait until the set of active conjectures is empty and then move *all* failed conjectures back to the active set, provided that at least one new lemma was proved since last attempting the conjecture. This guarantees termination.

We have experimented with more elaborate heuristics in this step, eagerly adding failed conjectures back. These heuristics can help in certain examples, but so far none have been sufficiently general. Perhaps surprisingly, the simple method described above works well for all examples in this article. More sophisticated heuristics are further work.

3 Examples

This section gives examples of successful proofs and their related theory explorations, as well as an example showing some current limitations of our approach.

3.1 Rotating the length a of list

This simple property of the `rotate` function is surprisingly difficult to prove[3]:

```
prop_rotate xs = rotate (length xs) xs =:= xs
```

The `rotate` function takes a natural number `n` and returns the list resulting from removing the `n` first elements and appending them to the end. Rotating a list by its length returns the original list. Although this property is very simple to state it is surprisingly hard to prove by mathematical induction, as it requires a generalised version to be proved, which implies `prop_rotate`. This generalisation itself can be proved by induction.

Given the standard definitions of append, length and Peano numbers with successor `S` and zero `Z`, and the below definition of `rotate`, HipSpec finds and proves such a generalisation, and uses it to prove `prop_rotate`:

```
rotate Z     xs     = xs
rotate (S n) []     = []
rotate (S n) (x:xs) = rotate n (xs ++ [x])
```

[3] Here, `=:=` is HipSpec's notation for equality.

The lemmas for which HipSpec needed induction are in Figure 2.2. Lemma (8) is the required generalisation, from which HipSpec proves `prop_rotate`, which follows as a special case when `ys` is the empty list. Notice that lemma (8) itself requires lemmas (1) and (2). A number of additional lemmas are also discovered, which are not of use in this particular proof, but could well be useful in other proofs. The whole process of theory exploration and the proof of `prop_rotate` took 17 seconds, with less than a second spent in QuickSpec and the rest of the time spent in various proofs of the generated equations.

No	Conjecture		By
(1)	`xs++[]`	`= xs`	
(2)	`(xs++ys)++zs`	`= xs++(ys++zs)`	
(3)	`rotate n (rotate m xs)`	`= rotate m (rotate n xs)`	
(4)	`rotate (S n) (rotate m xs)`	`= rotate (S m) (rotate n xs)`	(3)
(5)	`rotate n [x]`	`= [x]`	
(6)	`length (xs++ys)`	`= length (ys++xs)`	
(7)	`length (rotate n xs)`	`= length xs`	(6)
(8)	`rotate (length xs) (xs++ys)`	`= ys++xs`	(1,2)
(9)	`rotate (length xs) xs`	`= xs`	(8)

Figure 2.2: Properties generated and proved about the theory of lists with `++`, `rotate`, and `length`. The third column shows which lemmas were used.

As this proof requires both generalisation and lemma discovery it was identified in 2005 as an automated reasoning challenge beyond the capabilities of state-of-the-art reasoning systems [Bundy et al., 2005, p. 77]. We are not aware of any other theorem provers which prove this theorem fully automatically, without the help of user-supplied lemmas.

3.2 Nicomachus' Theorem

Using Peano arithmetic, with standard definitions of addition and multiplication recursively on the first argument, we will try to get HipSpec to prove Nicomachus' Theorem. This states that the sum of the n first cubes is the nth triangle number squared:

$$\sum_{k=1}^{n} k^3 = \left(\sum_{k=1}^{n} k \right)^2 .$$

We define two functions: `tri` calculates triangle numbers and `cubes n` calculates the sum of the first `n` cubes.

```
tri Z       = Z
tri (S n)   = tri n + S n
cubes Z     = Z
cubes (S n) = cubes n + (S n*S n*S n)
```

Using these definitions, Nicomachus' theorem is stated as follows:

```
prop_Nicomachus x = cubes x =:= tri x * tri x
```

When HipSpec is given the definitions of plus, multiplication, `tri` and `cubes`, it generates and proves (by induction) the properties listed in Figure 2.3 below, which takes 10 seconds. The properties are listed in the order they were proved.

No	Conjecture	Lemmas used	Induction on
(1)	`x+y = y+x`		x, y
(2)	`x+(y+z) = (y+x)+z`	(1)	z
(3)	`x*y = y*x`	(2)	x, y
(4)	`x*(y*z) = (y*x)*z`	(1), (2), (3)	x, y
(5)	`x*(y+y) = y*(x+x)`	(1), (2), (3), (4)	y
(6)	`(x*y)+(x*z) = x*(y+z)`	(1), (2), (3)	z
(7)	`tri x*(y+y) = (x*y)*S x`	(1), (2), (3), (4), (6)	x
(8)	`tri x+tri x = x+(x*x)`	(1), (2), (3)	x
(9)	`tri x*tri x = cubes x`	(1), (2), (3), (6), (8)	x

Figure 2.3: Properties proved about the theory with natural number addition, multiplication, triangle numbers (`tri`) and sum of cubes (`cubes`).

In (8) the well-known identity $\sum_{k=1}^{n} k = n(n+1)/2$ is proved, using previously proved lemmas. From this lemma HipSpec proves Nicomachus' Theorem in (9). Due to the order in which HipSpec ends up proving the conjectures in this example, some unnecessary lemmas are included in figure 2.3, e.g. (5) and (7).

3.3 Insertion sort produces a sorted list

Currently, QuickSpec can only generate equational lemmas. To prove that, for example, insertion sort produces a sorted list requires conditional lemmas. We state this property as:

```
prop_sorted xs = sorted (isort xs) =:= True~
```

In order to prove `prop_sorted` we need the conditional lemma `sorted xs ==> sorted (insert x xs)`, where `insert` is the sorted list

insertion function used by `isort`, but HipSpec only can only discover and prove the somewhat peculiar equations (lemmas *1-4*) in Figure 2.4. HipSpec also discovers, but fails to prove, some additional properties (conjectures *5-9*). For example, property (*5*), which states that `insert` is commutative in its first argument. These equations are not proved because they require conditional lemmas.

Although not proved, QuickSpec has tested these equations and not found a counterexample. Hence, even a failed proof attempt may at least give some insight into the properties of the program. The runtime for this example was 8 seconds.

No	Conjecture	
(*1*)	`x<=x`	`= True`
(*2*)	`x<=S x`	`= True`
(*3*)	`S x<=x`	`= False`
(*4*)	`insert y (x:[])`	`= insert x (y:[])`
(*5*)	`insert x (insert y xs)`	`= insert y (insert x xs)`
(*6*)	`sorted (insert x xs)`	`= sorted xs`
(*7*)	`isort (insert x xs)`	`= isort (x:xs)`
(*8*)	`sorted (isort xs)`	`= True`
(*9*)	`isort (isort xs)`	`= isort xs`

Figure 2.4: Results for the theory of insertion sort. Properties *1-4* were proved, while properties *5-9* were not, as they require conditional lemmas.

4 Evaluation

HipSpec has two modes of use. Firstly, it can be used as an automated induction to prove user-given conjectures using theory exploration to find necessary lemmas. In this case, the individual lemmas that are discovered in the background and used in the proofs are of less interest for the user, since the focus is on proving the user supplied properties automatically. Theory exploration is treated more like a black box.

Secondly, HipSpec can be used in a more speculative manner, as a standalone theory exploration system. In this case, the user expects HipSpec to discover and prove a set of basic equational properties about the given program. Here it becomes important not to swamp the user with trivial or overly complicated equations. Rather, we wish to present the user with a concise set of elegant equations summarising the main properties, much like the libraries in proof assistants such as Isabelle. The hope is that these may be useful in later interactive reasoning or as an algebraic specification of the program.

The examples come from the theorem proving literature and assume terminating functions over total values. We used Z3 [de Moura and Bjørner, 2008] as a backend for HipSpec in these experiments. As the program is translated to a first order theory, we did not use any of Z3's built-in theories or decision procedures, but we did exploit its support for types and constructor functions. The source code for HipSpec and all experimental results are available online [Claessen et al., 2013a,b].

4.1 HipSpec as a Theorem Prover

HipSpec was evaluated on two test suites from the inductive theorem proving literature. The test suites consist of conjectures about natural numbers, lists and binary trees. As they feature a large number of unrelated functions, HipSpec was run separately for each property. This reduces the number of generated equations because HipSpec will ignore any function that is not (directly or indirectly) reachable from the property. It also means that HipSpec cannot use already-proved properties from the test suite to prove later ones. Thus, the order of the properties in the test suite does not matter: they are proved independently.

HipSpec was configured to give a timeout of 1 second for each individual proof obligation sent to the prover, and to allow induction on up to two variables simultaneously using one-step structural induction.

Test Suite A consists of 85 conjectures with both first- and higher-order functions about lists, natural numbers and binary trees [Johansson et al., 2010a]. These were originally formalised for the IsaPlanner system in Isabelle's HOL and have since been translated into other formalisms to compare the Zeno and ACL2 Sedan provers [Chamarthi et al., 2011, Sonnex et al., 2011] and the Dafny system [Leino, 2010]. As these systems use different logics we note that the functions are not defined in exactly the same way in the different experiments. This test suite was originally designed for evaluating IsaPlanner's rippling heuristic in the presence of if- and case-expressions, which are expressed as higher-order functions in Isabelle, and cause trouble for IsaPlanner's syntax-based rippling heuristic. Hence, from a lemma discovery point of view, many proofs are rather easy: 67 theorems can be proved without extra lemmas, and 12 do not require induction. The results for the different provers on the 85 conjectures are summarised below:

HipSpec	Zeno	ACL2s	IsaPlanner	Dafny
80	82	74	47	45

HipSpec performs well, with the majority of failures being due to proofs requiring conditional lemmas, as HipSpec only is able to generate equations. For one property (number 81), we had to configure HipSpec to use induction on three variables; this is counted as a success in the table above. Zeno performs best, failing only on three examples, two fewer than HipSpec. However, HipSpec can prove two theorems that Zeno cannot:

```
rev (drop i xs) = take (len xs-i) (rev xs)
rev (take i xs) = drop (len xs-i) (rev xs)
```

Test Suite B consists of 50 theorems about lists and natural numbers and was previously used to demonstrate proof-critics in the CLAM prover [Ireland and Bundy, 1995], which is unfortunately no longer maintained. As opposed to Test Suite A, most theorems here do require auxiliary lemmas, generalisations, case-splits or non-standard inductions. CLAM proves 41 of the 50 theorems fully automatically. The remaining 9 theorems were proved interactively. They require generalisation (including the `rev` example from §1 and the `rotate` example from §3.1), for which CLAM needed the help of some user-supplied lemmas. Again, HipSpec was not given any auxiliary lemmas. Fully automatically, it proved 44 theorems, including 6 of the 9 theorems which CLAM proved with the help of user-supplied lemmas.

We managed to prove 3 further theorems (properties 33–35) by adjusting HipSpec's settings. These three properties concern accumulating versions of multiplication, factorial and exponentiation. Because we are using Peano arithmetic, these functions return large results, and the testing phase used too much memory: we supplied a flag that causes QuickSpec to compare results up to some size bound, so results that are too large will be considered equal. There were also too many conjectures, so we added a flag to limit the *size* of the generated terms. We did *not* have to give any lemmas by hand. In total, HipSpec proved 47 theorems, including the 9 for which CLAM needed user-supplied lemmas.

We also tested Zeno on these examples: it can prove 21, but not any of the ones requiring complex generalisations.

Finally, we remark that the bottom-up approach taken by HipSpec is naturally a bit slower than IsaPlanner and Zeno, which typically perform proofs in less than a second. Most successful proof attempts are very fast, with the long runtimes arising from cases with a lot of failed proof attempts.

For test suite A, all properties required less than a minute on a normal desktop computer [Claessen et al., 2013b]. The vast majority required less than 15 seconds, and most 1–2 seconds. For test suite B, the 44 successful properties required at most 15 seconds, most of them 1–2 seconds. The three properties for which we needed to tweak the settings ranged from 30 seconds to 40 minutes. Of the three failed properties, two took about five minutes before giving up, the third 8 seconds.

As mentioned, HipSpec may also discover some superfluous lemmas not strictly required for the proof of the user-stated property. In these examples, there are very few such lemmas and the theorem prover's performance was not notably affected by these being added to the theory.

4.2 HipSpec as a Theory Exploration System

In these experiments HipSpec is given a program as an input, without any user-properties stated. The aim is to present the user with a concise set of equational properties that have been discovered and proved. We exploit the pruning algorithm already implemented in QuickSpec to achieve this. QuickSpec was originally built as a standalone system for suggesting algebraic specifications of programs using testing. When used on its own, it prunes the many equations it generates by heuristically ordering them and removing those that trivially follow from previous ones. We refer to Paper 1 for a detailed description of this pruning algorithm. When HipSpec is used in theory exploration mode, it first attempts to prove as many conjectures as we can, just as in the theorem-proving mode. Then it takes the set of the conjectures that it proved, or that trivially follow from what it proved, and applies the pruning algorithm to this set. As a result, HipSpec often produces a smaller and more concise set of lemmas than it does when used in theorem-prover mode. The final list of equations does not depend on what order we proved things in, or on what needed induction, only on what the theory implies.

We have applied HipSpec to some simple theories from the theory exploration literature [Johansson et al., 2011, Montano-Rivas et al., 2012], one about natural numbers, with `+` and `*`, and three small theories about lists: 1) `append`, `reverse` and `length`, 2) `append`, `reverse` and `map` and 3) `append`, `foldl` and `foldr`. The theorems produced are presented in Figure 2.5. HipSpec generates these theorems much faster than IsaCoSy and IsaScheme: it takes only between 6-12 seconds for each theory (full results available online [Claessen et al., 2013b]), while

IsaCoSy and IsaScheme may require hours. We expect this to be due to the congruence closure reasoning of QuickSpec, which reduces the search space and integrates counterexample checking in the term generation phase.

Natural Numbers

N1.	`x+y = y+x`	*N6.**	`x*(y*z) = y*(x*z)`
N2.	`x*y = y*x`	*N7.*	`x+S y = S (x+y)`
N3.	`x+Z = x`	*N8.*	`x*S y = x+(x*y)`
N4.	`x*Z = Z`	*N9.**	`x*(y+y) = y*(x+x)`
*N5.**	`x+(y+z) = y+(x+z)`	*N10.*	`(x*y)+(x*z) = x*(y+z)`

Lists

L1.	`xs++[] = xs`
L2.	`(xs++ys)++zs = xs++(ys++zs)`
*L3.**	`length (xs++ys) = length (ys++xs)`
L4.	`length (rev xs) = length xs`
L5.	`rev (rev xs) = xs`
L6.	`rev xs++rev ys = rev (ys++xs)`
L7.	`map f xs++map f ys = map f (xs++ys)`
L8.	`map f (rev xs) = rev (map f xs)`
L9.	`foldl f (foldl f x xs) ys = foldl f x (xs++ys)`
L10.	`foldr f (foldr f x xs) ys = foldr f x (ys++xs)`

Figure 2.5: Theory exploration results: theorems generated by HipSpec. Theorems marked by $*$ were not in Isabelle's library.

We also perform the same precision-recall analysis as in Johansson et al. [2011] and Montano-Rivas et al. [2012] to assess the quality of the generated theories using Isabelle's libraries[4] as reference. This experiment assumes that the Isabelle library is so well-designed that it contains exactly all interesting properties and nothing more. The results are summarised in Table 2.1, where *recall* measures how many of the theorems in the library were also produced by HipSpec, and *precision* measures how many of the theorems HipSpec produced were also in the library, i.e. how well it avoids producing "superfluous" theorems.

HipSpec performs very well: for the lists, it generates all theorems in Isabelle's library, plus theorem *L3* in Figure 2.5, which is the closest we can get to the lemma `length (xs ++ ys) = length xs + length ys` since we did not include the `+` operator in the program. For the natural numbers, HipSpec fails to generate three of the library theorems:

[4] http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/library/HOL/

the standard formulations of associativity are missing (instead HipSpec generates two variants in theorems *N5* and *N6*) and the theorem `S x + y = x + S y` is excluded. However, all three can be trivially derived by equational reasoning from the theorems HipSpec does produce.

	HipSpec	IsaCoSy	IsaScheme	Isabelle
#Thms Naturals	10	16	16*	12
Precision	80%	63%	100%*	-
Recall	73%	83%	46%*	-
#Thms Lists	10	24	13	9
Precision	90%	38%	70%	-
Recall	100%	100%	100%	-

Table 2.1: Theory Exploration results. Note that IsaScheme was evaluated on a natural number theory also including exponentiation [Montano-Rivas et al., 2012].

5 Related Work

Inductive theories do not allow cut-elimination and are thus undecidable. In practice, this means that auxiliary lemmas (themselves requiring an inductive proof) may be required to complete a proof. Inductive theorem provers which support some form of automated lemma discovery, such as ACL2's induction tactic [Chamarthi et al., 2011], CLAM [Ireland and Bundy, 1995], IsaPlanner [Dixon and Fleuriot, 2004] and Zeno [Sonnex et al., 2011], use a *top-down* approach by which lemmas are discovered from failed proof-attempts. HipSpec differ from all of these in its *bottom-up* theory exploration approach. HipSpec automatically tries to discover a background theory for the relevant functions, building up something like the human-created lemma libraries available for interactive provers such as Isabelle [Nipkow et al., 2002] or ACL2 [Kaufmann et al., 2000]. Experimental evaluation shows that HipSpec's bottom-up approach compares well in terms of finding the right lemmas. Some types of lemmas are difficult to discover in the top-down approach, for instance the generalised version needed to prove the theorem `rev xs = qrev xs []`, and many other similar theorems featuring accumulator variables. While the CLAM system could discover the `rev`/`qrev` generalisation given some other basic lemmas, HipSpec discovers it all automatically. Zeno, IsaPlanner and ACL2 do not support this type of lemma discovery at all, and thus fail on theorems of this

kind. In HipSpec, there is always a risk of discovering extra irrelevant lemmas too. However, these may perhaps be useful in other proofs.

Both CLAM and IsaPlanner are based on the rippling heuristic for guiding rewriting of the step-case towards the inductive hypothesis. The advantage of rippling is that it guarantees termination of rewriting, and that rewrite rules may be used both ways around if need be. Rippling is a syntax-based heuristic, which may cause problems for instance on conjectures where a lot of case-analysis is required, as highlighted by Test Suite A in §4 where HipSpec, Zeno and ACL2 performed better than the rippling-based IsaPlanner. HipSpec relies on an off-the-shelf prover as backend which has no termination guarantee like rippling-based provers. Instead termination is enforced by using a timeout, which means that there is a risk of missing proofs which just take a little bit too long. When special-purpose rippling-based provers fail, the user may inspect the final proof state to see where the proof got stuck. HipSpec cannot currently.

While most other provers have some form of built-in rewriting tactics, HipSpec and the program verifier Dafny [Leino, 2010] instead send proof obligations to external automated provers. Like HipSpec, Dafny applies induction on the meta-level and passes the resulting proof obligations to the theorem prover Z3, which was also used as a backend for HipSpec in the experiments in this article. Dafny does not, however, support automated lemma discovery, so auxiliary lemmas must be supplied by the user. The obvious advantage is that off-the-shelf automated provers are often very fast and powerful. However, as the provers are treated as black boxes we do not get a readable proof, or any information if a proof fails. IsaPlanner checks proof steps in Isabelle and can produce readable output of complete or partial proofs. Zeno can output proofs in Isabelle format, which can then be re-checked in the proof assistant, ensuring correctness. Readable and checkable proofs are further work in HipSpec.

HipSpec is the only system which can be used both as an inductive theorem prover and as a theory exploration system. The IsaCoSy and IsaScheme theory explorers were developed for automating the creation of lemma libraries for inductive theories in Isabelle [Johansson et al., 2011, Montano-Rivas et al., 2012]. Both systems use IsaPlanner to prove conjectures that pass counterexample checking, but differ in the heuristics they use to generate conjectures. Experiments in which the outputs of IsaCoSy were *manually* fed back to IsaPlanner have been successfully performed [Johansson et al., 2010b]. However, neither is fully integrated with the theorem prover: IsaPlanner cannot call

either of these *automatically* while proving user-given properties, In contrast, HipSpec is fully automatic. Both IsaCoSy and IsaScheme are considerably slower than HipSpec, although all three systems produce similar sets of lemmas.

6 Conclusion and Further Work

HipSpec is an automated inductive theorem prover and a theory exploration system. It takes a novel bottom-up approach to lemma discovery by using theory exploration to first build a richer background theory in which user-given properties are proved. In experimental evaluation, HipSpec performs very well in comparison with other systems: in particular, it succeeds in proving theorems about tail-recursive functions that require generalisations, which no other system can prove fully automatically without user-supplied lemmas. HipSpec also performs very well as a standalone theory exploration system, producing sets of lemmas with high precision and recall when compared to Isabelle's libraries. Furthermore, it does so in seconds rather than hours like previous systems.

Ultimately, we would like to use HipSpec in a tool for automatically proving properties of Haskell programs, making it usable by "normal" programmers, much like the popular QuickCheck tool [Claessen and Hughes, 2000]. In order to extend HipSpec to the full Haskell language we need to add support also for infinite and lazy data-structures and non-terminating functions in QuickSpec and in HipSpec's property language. The Haskell-to-FOL translation system HALO [Vytiniotis et al., 2013] already supports this, and Hip supports co-inductive reasoning and fixpoint induction. The theory-exploration machinery does however need to be extended to record which lemmas hold for all values of a type (including partial ones) and which ones only hold for completely-defined total values.

Another area of further work is providing user feedback from failed proofs, and producing checkable proofs. It could be interesting to experiment with a different prover backend, from which information about failed proof attempts can be reclaimed, rather than treating the prover as a black box.

3

Sort It Out with Monotonicity: Translating between Many-Sorted and Unsorted First-Order Logic

This paper was published at CADE 2011 in Wrocław. The CADE version contained a flawed proof of Theorem 1, which has now been fixed; thanks to Jasmin Blanchette and Andrei Popescu for pointing it out.

Chapter 3

Sort It Out with Monotonicity: Translating between Many-Sorted and Unsorted First-Order Logic

Koen Claessen Ann Lillieström
Nicholas Smallbone

Abstract

We present a novel analysis for sorted logic, which determines if a given sort is *monotonic*. The domain of a monotonic sort can always be extended with an extra element. We use this analysis to significantly improve well-known translations between unsorted and many-sorted logic, making use of the fact that it is cheaper to translate monotonic sorts than non-monotonic sorts. Many interesting problems are more naturally expressed in many-sorted first-order logic than in unsorted logic, but most existing highly-efficient automated theorem provers solve problems only in unsorted logic. Conversely, some reasoning tools, for example model finders, can make good use of sort-information in a problem, but most problems today are formulated in unsorted logic. This situation motivates translations in both ways between many-sorted and unsorted problems. We present the monotonicity analysis and its implementation in our tool *Monotonox*, and also show experimental results on the TPTP benchmark library.

1 Introduction

Many problems are more naturally expressed in many-sorted first-order logic than in unsorted logic, even though their expressive power is equivalent. However, none of the major automated theorem provers for first-order logic can deal with sorts. Most problems in first-order logic are therefore expressed in unsorted logic.[1] However, some automated reasoning tools (such as model finders) could greatly benefit from sort information in problems.

This situation motivates the need for translations between sorted and unsorted first-order logic: (1) users want to express their problems in sorted logic, whereas many tools only accept unsorted logic; (2) some tool developers want to work with sorted logic, whereas the input problems are mostly expressed in unsorted logic. For example, a model finder for a sorted logic has more freedom than for an unsorted logic: it can find domains of different sizes for different sorts, and apply symmetry reduction for each sort separately.

In this paper, we describe automated ways of translating back and forth between many-sorted and unsorted first-order logic. We use a novel *monotonicity* analysis to improve on well-known existing translations. In short, a sort is *monotonic* in a problem if, for any model, the domain of that sort can be made larger without affecting satisfiability. The result of the translation for monotonic sorts turns out to be much simpler than for non-monotonic sorts. The monotonicity analysis and the translations are implemented in a tool called Monotonox.

To explain the problem we solve, and how monotonicity helps us, we will use the following running example.

Example 1 (monkey village). There exists a village of monkeys, with a supply of bananas. Every monkey must have at least two bananas to eat. A banana can not be shared among two monkeys. To model this situation we introduce two sorts, one of monkeys and one of bananas. We need a predicate $\mathsf{owns} \in \mathit{monkey} \times \mathit{banana} \to \mathrm{o}$ that says which monkey owns each banana, and Skolem functions banana_1 and $\mathsf{banana}_2 \in \mathit{monkey} \to \mathit{banana}$ to witness the fact that each monkey has

[1] Indeed, only recently was a collection of many-sorted first-order problems added to the TPTP [Sutcliffe, 2009].

two bananas. We use the following four axioms:

$$\forall M \in \mathit{monkey}.\, \mathsf{owns}(M, \mathsf{banana}_1(M)) \tag{1}$$

$$\forall M \in \mathit{monkey}.\, \mathsf{owns}(M, \mathsf{banana}_2(M)) \tag{2}$$

$$\forall M \in \mathit{monkey}.\, \mathsf{banana}_1(M) \neq \mathsf{banana}_2(M) \tag{3}$$

$$\forall M_1 \in \mathit{monkey}, M_2 \in \mathit{monkey}, \\ B \in \mathit{banana}.\, (\mathsf{owns}(M_1, B) \wedge \mathsf{owns}(M_2, B) \Longrightarrow M_1 = M_2) \tag{4}$$

We use a simple but standard many-sorted first-order logic, in which sorts α have a non-empty domain $D(\alpha)$, all symbols have exactly one sort, there are no subsorts, and equality is only allowed between terms of the same sort.

If we want to use a standard reasoning tool for unsorted logic (for example a model finder) to reason about the monkey village, we need to translate the problem into unsorted logic. Automated reasoning folklore [Wick and McCune, 1989] suggests three alternatives:

Sort predicates The most commonly used method is to introduce a new unary predicate p_α for every sort α that is used in the sorted formula [Enderton, 2001, §4.3]. All quantification over a sort α is translated into unsorted quantification bounded by the predicate p_α. Furthermore, for each function or constant symbol in the problem, we have to introduce an extra axiom stating the result sort of that symbol. For example, the first axiom of example 1 translates to

$$\forall M.\, (\mathsf{p}_{\mathit{monkey}}(M) \rightarrow \mathsf{owns}(M, \mathsf{banana}_1(M))$$

and we have to add axioms like $\forall M.\, \mathsf{p}_{\mathit{banana}}(\mathsf{banana}_1(M))$ for each function symbol. Moreover, to rule out the possibility of empty sorts, we sometimes need to introduce axioms of the form $\exists X.\, \mathsf{p}_\alpha(X)$.

Although conceptually simple, this translation introduces a lot of clutter which affects most theorem provers negatively: one extra predicate symbol for each sort, one axiom for each function symbol, and one extra literal for each variable in each clause. (The theorem prover SPASS [Weidenbach et al., 2002] deals with these symbols and literals specially for this reason.)

Sort functions An alternative translation introduces a new function symbol f_α for each sort α. The translation applies f_α to any subterm of sort α in the sorted problem. The aim is to have the image of f_α in the unsorted problem be the domain of α in the sorted problem; f_α thus

maps any arbitrary domain element into a member of the sort α. For example, using sort functions, the first axiom of example 1 translates to

$$\forall M.\, \mathsf{owns}(\mathsf{f}_{monkey}(M), \mathsf{f}_{banana}(\mathsf{banana}_1(\mathsf{f}_{monkey}(M))))$$

No additional axioms are needed. Thus, this translation introduces a lot less clutter than the previous translation. Still, the performance of theorem provers is affected negatively, and it depends on the theorem prover as well as the problem which translation works best in practice.

Sort erasure The translation which introduces least clutter of all simply *erases* all sort information from a sorted problem, resulting in an unsorted problem. However, while the two earlier mentioned translations preserve satisfiability of the problems, sort erasure does not, and is in fact unsound. Let us see what happens to the monkey village example. Erasing all the sorts, we get

$$\begin{gathered}
\forall M.\, \mathsf{owns}(M, \mathsf{banana}_1(M)) \\
\forall M.\, \mathsf{owns}(M, \mathsf{banana}_2(M)) \\
\forall M.\, \mathsf{banana}_1(M) \neq \mathsf{banana}_2(M) \\
\forall M_1, M_2, B.\, (\mathsf{owns}(M_1, B) \wedge \mathsf{owns}(M_2, B) \rightarrow M_1 = M_2)
\end{gathered}$$

This new problem *has no finite model*, even though the sorted problem does! The reason is that, if the domain we choose has finite size k, we are forced to have k monkeys and k bananas. But a village of k monkeys must have $2k$ bananas, so this is impossible unless the domain is infinite (or empty, which we disallow). So, sort erasure does not preserve finite satisfiability, as shown by the example. In fact, it does not even preserve satisfiability.

Related work The choice seems to be between translations that are sound, but introduce clutter that gets in the way of the theorem prover, and a translation that introduces no clutter but is unsound. Automated theorem provers for unsorted first-order logic have been used to reason about formulae in Isabelle [Meng and Paulson, 2008, Paulson, 1999]. The tools apply sort erasure, and investigate the proof to see if it made use of unsound reasoning. If that happens they can use a sound but inefficient translation as a fall-back. A similar project using AgdaLight [Abel et al., 2005] uses sort erasure but, following Wick and McCune [1989], proposes that the theorem prover be restricted to not use certain

rules (i.e. paramodulation on variables), leading to a sound (but possibly incomplete) proof procedure.

Monotonicity has been studied for higher-order logic [Blanchette and Krauss, 2011] to help with pruning the search space when model finding. While the intention there is the same as ours and there are similarities between the approaches, the difference in logics changes the problem dramatically. For example, we infer that any formula without $=$ is monotonic, which is not true in higher-order logic. Monotonicity is also related to the ideas of stable infiniteness and smoothness [Ranise et al., 2005] in combining theories in SMT; it would be interesting to investigate this link further.

This paper We give an alternative to choosing between clutter and unsoundness. We propose an analysis that indicates which sorts are safe to erase, leaving ideally only a few sorts left that need to be translated using one of the first two methods.

The problem with sort erasure is that it forces all sorts to use the same domain. If the domains all had the same size to start with, there is no problem. But if the sorted formula only has models where some domains have different sizes than others, the sort erasure makes the formula unsatisfiable. We formulate this observation in the following lemma:

Lemma 1. *The following statements about a many-sorted first-order formula* φ *are equivalent:*

1. *There is an unsorted model with domain* D *for the sort-erased version of* φ.
2. *There is a model of* φ *where the size of each domain is* $|\mathsf{D}|$.

Proof. (sketch) The interesting case relies on the observation that, if there is a sorted model in which all domains have the same size, then there is also a model in which all the domains are identical, from which it is trivial to construct an unsorted model. □

Our main contribution is a *monotonicity inference* calculus that identifies the so-called *monotonic* sorts in a problem. If all sorts in a satisfiable sorted problem are monotonic, then it is guaranteed that there will always be models for which all domains have the same size, in which case sort erasure is sound. The sorts that cannot be shown monotonic will have to be made monotonic first by introducing sort predicates or functions, but for these sorts only.

2 Monotonicity Calculus for First-Order Logic

Monotonox exploits *monotonicity* in the formula we are translating to produce a more efficient translation than the naive one. The purpose of this section is to explain what monotonicity is and how to infer it in a formula; section 3 explains how we use this information in Monotonox.

Before tackling monotonicity in a sorted setting, we first describe it in an unsorted one. We do this just because the notation gets in the way of the ideas when we have sorts.

2.1 Monotonicity in an Unsorted Setting

We start straight away with the definition of monotonicity. Monotonicity is a semantic property rather than a syntactic property of the formula.

Definition 1 (monotonicity, unsorted)**.** An unsorted formula φ is *monotonic* if, for all $n \in \mathbb{N}$, whenever φ is satisfiable over domains of size n, it is also satisfiable over domains of size $n + 1$.

An immediate consequence is that if a monotonic formula is satisfiable over a finite domain, it is also satisfiable over all bigger finite domains.

Remark 1. Several common classes of formulae are monotonic:

- Any unsatisfiable formula is monotonic because it trivially satisfies our definition. The same goes for any formula that has no finite models.
- Any valid formula is monotonic because it has a model no matter what the domain is.
- A formula that does not use $=$ is monotonic, as we will see later.

What about a non-monotonic formula? The simplest example is $\forall X, Y. X = Y$, which is satisfied if the domain contains a single element but not if it contains two. We will see later that equality is the single source of non-monotonicity in formulae.

Monotonicity allows us to take a model of a formula and get from it a model over a bigger domain. Although it is not obvious from our definition, this is even the case if we want to get an infinite model. We stay out of technical deep water here by restricting ourselves to countably infinite models.

Lemma 2 (monotonicity extends to countable domains)**.** *φ is monotonic iff, for every pair of domains D and D' such that $|\mathsf{D}| \leqslant |\mathsf{D}'|$, if φ is satisfiable over D and D' is countable then φ is satisfiable over D'.*

Proof. (sketch) If φ is monotonic and has a finite model then it has models of unbounded size; by compactness it has an infinite model. □

Monotonicity is not decidable We can see from remark 1 that monotonicity is related to satisfiability, so we should not expect it to be decidable. Indeed it is not.[2] This does not mean we should give up on inferring monotonicity, just that we cannot *always* infer that a formula is monotonic. The calculi we present later only answer "yes" if a formula is monotonic but may answer "no" for a monotonic formula too.

2.2 Monotonicity in a Many-Sorted Setting

Everything above generalises to sorted formulae, with the complication that we now have to talk about a formula being monotonic in a particular sort. Informally, φ is monotonic in the sort α if, given a model of φ, we can add elements to the domain of α while preserving satisfiability.

We use the notation $\mathsf{D}(\alpha)$ for the domain of sort α. The formal definition mimics the one from the last section:

Definition 2 (monotonicity, sorted)**.** A sorted formula φ is *monotonic in the sort* α if, whenever φ is satisfiable over D, and we are given D' such that

- $|\mathsf{D}(\alpha)|$ is finite, and $|\mathsf{D}(\alpha)| + 1 = |\mathsf{D}'(\alpha)|$, and
- $\mathsf{D}'(\beta) = \mathsf{D}(\beta)$ for all $\beta \neq \alpha$,

then φ is satisfiable over D'.

Once again, we only consider taking a finite domain and adding a single element to it. The lemma from the last section holds:

[2] The proof works by encoding a given Turing machine by a formula that has a finite model of size k iff the Turing machine halts in exactly k steps. Thus if the Turing machine halts then the formula has a finite model at exactly one domain size and is therefore not monotonic; if the Turing machine does not halt then the formula is finitely unsatisfiable and therefore monotonic.

Lemma 3 (monotonicity extends to countable domains (sorted))**.** φ *is monotonic in* α *iff, whenever* φ *is satisfiable over* D*, and we are given* D' *such that*

- $|\mathsf{D}'(\alpha)| \geqslant |\mathsf{D}(\alpha)|$,
- $\mathsf{D}'(\beta) = \mathsf{D}(\beta)$ *for all* $\beta \neq \alpha$*, and*
- $\mathsf{D}'(\beta)$ *is countable for all* β,

then φ *is satisfiable over* D'.

The key insight of Monotonox is that *sort erasure is safe if the formula is monotonic* in all sorts:

Theorem 1 (monotonic formulae preserve satisfiability under erasure)**.** *If* φ *is a many-sorted monotonic formula, then* φ *and its sort-erasure are equisatisfiable.*

Proof. By lemma 1, it is enough to show that from a model of φ we can construct a model where all domains are the same size. By lemma 3 we can do this by extending all the domains to match the size of the biggest domain, if the model is countable.

If the model is uncountable, first apply the Downward Löwenheim-Skolem theorem to get a countable model, then use the argument above. □

Remark 2. Notice that this construction preserves finite satisfiability, which is important when we are going to use a finite model finder on the problem.

Going back to our monkeys example, the formula is monotonic in the sort *banana* (you can always add a banana to the model) but not in the sort *monkey* (if we have k monkeys and $2k$ bananas, we may not add another monkey without first adding two bananas). In section 4 we will see that this means we only need to introduce a sort predicate for the sort *monkey*.

2.3 A Simple Calculus for Monotonicity Inference

We now present two calculi for inferring monotonicity of a formula. In both calculi we assume that the formula is in CNF.

Our first calculus is based on the key observation that any formula that does not use equality is monotonic. To see why, suppose we have

a model over domain D of a formula φ, and we want to add a new element to D while preserving satisfiability. We can do this by taking an existing domain element $e \in D$ and making the new element e' behave identically to e, so that for all unary predicates P, $P(e)$ is true iff $P(e')$ is true, and for all unary functions f, $f(e) = f(e')$, and similarly for predicates and functions of higher arities. If the formula does not use equality, e and e' cannot be distinguished. Thus, the addition of a new domain element preserves satisfiability of the formula.

On the other hand, with equality present, the addition of a new element to the domain may make a previously satisfiable formula unsatisfiable. For example, $\forall X, Y.\, X = Y$ has a model with domain size 1, but it is not satisfiable for any larger domain size. We cannot make the new domain element behave the same as the old domain element because equality can distinguish them.

However, not all occurrences of equality have this problem. The following examples of equality literals are all monotonic:

1. Negative equality (by increasing the size of the domain, more terms may become unequal but previously unequal terms will not become equal).

2. Equality where neither side is a variable (i.e. both sides are functions or constants, possibly with variable arguments). This is because, by using the strategy above for extending the domain with a new element, no function ever returns the new element, so the new element is never tested for equality.

3. Equality over a sort α is monotonic in any sort β different to α. (The satisfiability of $t_1 = t_2$, where t_1 and t_2 have sort α is unaffected by the addition of new elements to the domain of β).

Thus, the only problematic literal for monotonicity in the sort α is positive equality over α where either side of the equality is a variable.

Safe terms We call a term *safe* in a sort α if, whenever we add a new element to the domain of α, the term never evaluates to this element. If the terms occurring on each side of an equality literal are both safe, the satisfiability of the literal is unaffected by the addition of new domain elements. Since positive equality literals are the only possible sources of non-monotonicity, we can infer monotonicity of a formula by showing that all arguments of positive equality literals are safe. By the examples above, a term is safe in the sort α if it is not a variable, or it has a

sort different to α. The simple calculus exploits these facts with the following rules:

1. $\varphi_1 \vee \varphi_2$ is monotonic in α iff φ_1 and φ_2 are monotonic in α.
2. $\varphi_1 \wedge \varphi_2$ is monotonic in α iff φ_1 and φ_2 are monotonic in α.
3. Any non-equality literal is monotonic in any sort α.
4. $t_1 \neq t_2$ is monotonic in any sort α.
5. $t_1 = t_2$ is monotonic in α if t_1 and t_2 are safe in α, i.e., are not variables or are not of sort α.

Let us try out the simple calculus on the hungry monkeys in Example 1. The formula is monotonic in *monkey* iff all of its clauses are monotonic in *monkey*, and similarly for *banana*. Clauses (1) and (2) are monotonic in both sorts, because the clauses do not contain equality. (3) is monotonic in both sorts, because the clause does not contain positive equality. (4) is monotonic in *banana*, because there is no equality between *banana* elements. The calculus does not let us infer monotonicity of *monkey* in this clause, because of the occurrence of an equality literal with two variables of sort *monkey*. Thus, the formula is monotonic in *banana*, but not in *monkey*. This is consistent with our previous observation that we can add more *banana* elements without affecting satisfiability, but this is not the case for *monkey* elements.

2.4 Improved Calculus

There are many cases when our first calculus is not able to prove monotonicity. For example, suppose we change the problem so that some monkeys are not hungry and do not need bananas:

Example 2.

$$\forall M \in \mathit{monkey}.\, (\mathsf{hungry}(M) \Longrightarrow \mathsf{owns}(M, \mathsf{banana}_1(M))) \tag{5}$$

$$\forall M \in \mathit{monkey}.\, (\mathsf{hungry}(M) \Longrightarrow \mathsf{owns}(M, \mathsf{banana}_2(M))) \tag{6}$$

$$\forall M \in \mathit{monkey}.\, (\mathsf{hungry}(M) \Longrightarrow \mathsf{banana}_1(M) \neq \mathsf{banana}_2(M)) \tag{7}$$

$$\begin{aligned} &\forall M_1, M_2 \in \mathit{monkey}, B \in \mathit{banana}. \\ &\quad ((\mathsf{hungry}(M_1) \wedge \mathsf{hungry}(M_2) \wedge \mathsf{owns}(M_1, B) \wedge \mathsf{owns}(M_2, B) \\ &\qquad \Longrightarrow M_1 = M_2) \end{aligned} \tag{8}$$

It is not hard to see that, given a model of the axioms, we can always add an extra monkey, by making that monkey not be hungry. Thus, the above formula is monotonic in *monkey*. However, our simple calculus can not infer this, because of the use of positive equality between two variables of sort *monkey* in (8). In this section we remedy the problem by extending the calculus.

In the simple calculus, the strategy for extending a model while preserving finite satisfiability was to pick an existing element e in the domain, and let any new domain element "mimic" e. This strategy does not work for clause (8) in Example 2: if we happen to pick an e such that $\mathsf{hungry}(e)$ is true, then this strategy will add an extra hungry monkey to the domain, which does not preserve finite satisfiability. In our improved calculus we can make use of alternative strategies for extending the model, which allows us to infer monotonicity in cases such as this.

Extension rules In the improved calculus, we nominate some predicates to be *"true-extended"* and some to be *"false-extended"* in each sort α. If a predicate is neither true-extended nor false-extended, we say that it is *"copy-extended"*. When extending the model with a new domain element e', if a predicate P is true-extended, we make P return true whenever any of its arguments is e'; likewise if it is false-extended we make it return false if any of its arguments is e'. Copy-extended predicates behave as in the simple calculus.

Guard literals We say that a literal $P(\ldots)$ in a clause C *guards* an occurrence of a variable $X \in \alpha$ in C if X is one of the arguments of that literal and P is true-extended in α. Similarly, a literal $\neg P(\ldots)$ in C with X among its arguments guards occurrences of X in C if P is false-extended in α. We call the literal $P(\ldots)$ or $\neg P(\ldots)$ in this case a *guard literal*. The idea is that when X is instantiated with the new domain element, the guard literal is true, hence satisfiability of the clause is preserved. This allows us to infer that a clause involving positive equality between variables is monotonic, if those variables are guarded. For example, in the clause (8) in Example 2, the two variables M_1 and M_2 occurring in the equality literal are guarded by the predicate hungry, which we can make false for any new elements of sort *monkey* that we add.

Furthermore, $X \neq t$ guards X if t is not a variable: the clause $X \neq t \vee \varphi[X]$ is equivalent to $X \neq t \vee \varphi[t]$, in which X does not appear

unsafely.[3]

Contradictory extensions When considering formulae, things get more problematic: if we add an axiom

$$\forall M \in \mathit{monkey}.\, \mathsf{hungry}(M) \tag{9}$$

to the formula in Example 2, we cannot add non-hungry monkeys to the domain, so the problem is no longer monotonic in the sort *monkey*. For the clause (8) to be monotonic, M_1 and M_2 must be guarded, which means that the predicate hungry must be false-extended. But extending hungry with false will not preserve satisfiability of the clause (9).

The new extension rules thus require some caution. If a predicate P is false-extended, then any occurrence of a variable X in the literal $P(\ldots)$ needs guarding just like it does in an equality literal $X = t$. Likewise, if P is true-extended, any occurrence of a variable X in the literal $\neg P(\ldots)$ needs guarding. This is illustrated in Example 3:

Example 3.

$$\forall X.\, (P(X) \Longrightarrow X = t) \tag{10}$$
$$\forall X.\, (Q(X) \Longrightarrow P(X)) \tag{11}$$

(10) requires P to be false-extended, because the occurrence of X in the positive equality literal needs guarding. But if $P(X)$ is false whenever X is instantiated with a new domain element, then Q must be false-extended in order to satisfy (11).

An occurrence of a variable X is problematic if it occurs in a literal of one of the following forms:

- $X = t$ or $t = X$
- $P(\ldots, X, \ldots)$ where P is false-extended
- $\neg P(\ldots, X, \ldots)$ where P is true-extended

In that case, we need to guard X for the formula to be monotonic in X's sort.

The improved calculus infers monotonicity of a formula in α iff there is a consistent extension of predicates that guards all such variable occurrences.

[3] This even holds if X is a subterm of t.

2.5 Monotonicity Inference Rules of the Improved Calculus

Notation In the following, we shall use the abbreviation K to denote a function from predicates to the extension methods $\{\mathsf{true}, \mathsf{false}, \mathsf{copy}\}$. We call such a K a *context*. Furthermore, we use the notation $\mathsf{K} \rhd_\alpha \varphi$ to mean that φ is monotonic in the sort α, given the context K.

Formulae A formula φ is monotonic with context K in the sort α iff all of its clauses are monotonic with K in α:

$$\frac{\mathsf{K} \rhd_\alpha C_1 \quad \cdots \quad \mathsf{K} \rhd_\alpha C_n}{\mathsf{K} \rhd_\alpha C_1 \wedge \ldots \wedge C_n}$$

Clauses In the rule for clauses, we must also consider the set Γ of variables that are guarded in the clause. We write $\Gamma, \mathsf{K} \rhd_\alpha l$ if l is monotonic with K in α, given that the variables in Γ are guarded. A clause is monotonic with context K in the sort α if all of its literals are monotonic with K in α, given Γ:

$$\frac{\Gamma = \bigcup_{i=1}^{n} \mathsf{guarded}(\mathsf{K}, l_i) \qquad \Gamma, \mathsf{K} \rhd_\alpha l_1 \quad \cdots \quad \Gamma, \mathsf{K} \rhd_\alpha l_n}{\mathsf{K} \rhd_\alpha l_1 \vee \ldots \vee l_n}$$

where $\mathsf{guarded}(\mathsf{K}, l)$ is defined as

$\mathsf{guarded}(\mathsf{K}, P(t_1 \ldots t_n)) = \{X \mid X \in \{t_1 \ldots t_n\}, X \text{ is a variable}\}$
if $\mathsf{K}(P) = \mathsf{true}$,
$\mathsf{guarded}(\mathsf{K}, \neg P(t_1 \ldots t_n)) = \{X \mid X \in \{t_1 \ldots t_n\}, X \text{ is a variable}\}$
if $\mathsf{K}(P) = \mathsf{false}$,
$\mathsf{guarded}(\mathsf{K}, X \neq t) = \{X\}$ if X is a variable and t is not,
$\mathsf{guarded}(\mathsf{K}, l) = \emptyset$ otherwise.

Literals We have the following rules for monotonicity inference of literals:

$$\frac{}{\Gamma, \mathsf{K} \rhd_\alpha t \neq_\beta u} \ (1) \qquad \frac{\beta \neq \alpha}{\Gamma, \mathsf{K} \rhd_\alpha t =_\beta u} \ (2)$$

$$\frac{\mathsf{safe}(\Gamma, t, \alpha) \qquad \mathsf{safe}(\Gamma, u, \alpha)}{\Gamma, \mathsf{K} \rhd_\alpha t =_\alpha u} \ (3)$$

where

$$\mathsf{safe}(\Gamma, t, \alpha) = \begin{cases} t \in \Gamma & \text{if } t \text{ is a variable of sort } \alpha, \\ \mathsf{true} & \text{otherwise.} \end{cases}$$

(1) Negative equality is always monotonic. (2) Equality in a sort β is monotonic in any sort α that is different to β. (3) Equality between two terms is monotonic if the terms are non-variables, or are guarded in the clause.

$$\frac{\mathsf{safe}(\Gamma, t_1, \alpha) \ \cdots \ \mathsf{safe}(\Gamma, t_n, \alpha)}{\Gamma, K \rhd_\alpha P(t_1, \ldots, t_n)} \quad (4)$$

$$\frac{\mathsf{safe}(\Gamma, t_1, \alpha) \ \cdots \ \mathsf{safe}(\Gamma, t_n, \alpha)}{\Gamma, K \rhd_\alpha \neg P(t_1, \ldots, t_n)} \quad (5)$$

(4,5) A predicate literal is monotonic in α if all of its variable arguments of sort α are guarded in the clause in which the literal occurs.

$$\frac{K(P) \in \{\mathsf{true}, \mathsf{copy}\}}{\Gamma, K \rhd_\alpha P(t_1, \ldots, t_n)} \quad (6) \qquad \frac{K(P) \in \{\mathsf{false}, \mathsf{copy}\}}{\Gamma, K \rhd_\alpha \neg P(t_1, \ldots, t_n)} \quad (7)$$

(6) A positive occurrence of a predicate is monotonic if the predicate is true-extended or copy-extended. (7) A negative occurrence of a predicate is monotonic if the predicate is false-extended or copy-extended.

It is not immediately clear how to implement the above rules, since there is no obvious way to infer the context K. We see in section 3.1 that we can do this using a SAT-solver.

2.6 NP-completeness of the Improved Calculus

The improved calculus allows us to infer monotonicity in more cases. However, inferring monotonicity with it is NP-complete. We show NP-hardness by reducing CNF-SAT to a problem of inferring monotonicity in the calculus.

Given any propositional formula φ_{SAT} in CNF, we construct a formula φ_{MON} such that φ_{SAT} is satisfiable iff φ_{MON} is monotonic. The idea is that a context that makes φ_{SAT} monotonic corresponds to a satisfying assignment for φ_{SAT}.

For each positive literal l in φ_{SAT}, we introduce a unary predicate P_l in φ_{MON}. For negative literals $\neg l$, we define $P_{\neg l}(X)$ as $\neg P_l(X)$. We equip

φ_{MON} with a single constant c. We translate each clause $(l_1 \vee ... \vee l_n)$ of φ_{SAT} into the following clause in φ_{MON}:

$$\forall X.\, P_{l_1}(X) \vee ... \vee P_{l_n}(X) \vee X = c$$

Our calculus proves this clause monotonic exactly when our context extends at least one of $P_{l_1}, .., P_{l_n}$ by true. Thus if we find a context that makes φ_{MON} monotonic we may extract a satisfying assignment for φ_{SAT} by doing the following for each positive literal l of φ_{SAT}:

- If P_l is extended by true then let l be true.
- If P_l is extended by false then let l be false.
- If P_l is extended by copy then choose an arbitrary value for l.

The same method takes us from a satisfying assignment of φ_{SAT} to a context that makes φ_{MON} monotonic.

3 Monotonox: Sorted to Unsorted Logic and Back Again

We have implemented the monotonicity calculus as part of our tool Monotonox. This section first shows how the calculus is implemented and then how monotonicity is exploited in translating between sorted and unsorted first-order logic.

3.1 Monotonicity Inference with Monotonox

We show in this section how to use a SAT-solver to implement the monotonicity calculus. The use of a SAT-solver is a reasonable choice, as we have seen previously that monotonicity inference in our calculus is NP-hard.

We encode the problem of inferring monotonicity of a formula φ as a SAT-problem, where a satisfying assignment corresponds to a context in our calculus.

We construct for each predicate P in φ two literals, p_T and p_F. The idea is that if p_T is assigned true, then P should be true-extended. If p_F is assigned true, then P should be false-extended. If both p_T and p_F are assigned false, then P should be copy-extended. Our task is to construct a propositional formula with these literals, that is satisfiable iff φ is monotonic according to our calculus.

Formulae The SAT-encoding of a formula φ is the conjunction of SAT-encodings of the clauses of φ and the constraint that each predicate may not be extended by both true and false:

$$\mathsf{mono}((C_1 \wedge .. \wedge C_n), \alpha) = \bigwedge_{i=1}^{n} \mathsf{mono}(C_i, \alpha) \wedge \bigwedge_{P_i \in \varphi} \neg p_{i_F} \vee \neg p_{i_T}$$

Clauses The SAT-encoding of a clause C is the conjunction of SAT-encodings of the literals of C.

$$\mathsf{mono}((l_1 \vee .. \vee l_n), \alpha) = \bigwedge_{i=1}^{n} \mathsf{mono}((l_1 \vee .. \vee l_n), l_i, \alpha)$$

Literals The SAT-encoding of a literal may depend on the clause in which it occurs. In a positive equality literal, both of the terms must be safe. A negative equality literal is trivially monotonic. An occurrence of a predicate is monotonic if the predicate is extended in an appropriate way or its arguments are safe.

$$\mathsf{mono}(C, l, \alpha) = \begin{cases} \mathsf{safe}(C, t_1, \alpha) \wedge \mathsf{safe}(C, t_2, \alpha) & \text{if } l \text{ is } t_1 = t_2, \\ \mathsf{true} & \text{if } l \text{ is } t_1 \neq t_2, \\ \neg p_F \vee \bigwedge_{i=1}^{n} \mathsf{safe}(C, t_i, \alpha) & \text{if } l \text{ is } P(t_1, \ldots, t_n), \\ \neg p_T \vee \bigwedge_{i=1}^{n} \mathsf{safe}(C, t_i, \alpha) & \text{if } l \text{ is } \neg P(t_1, \ldots, t_n), \end{cases}$$

A term t is safe in a clause if it is not a variable of the sort considered for monotonicity, or it is guarded by any of the literals in the clause.

$$\mathsf{safe}((l_1 \vee .. \vee l_n), t, \alpha) = \begin{cases} \bigvee_{i=1}^{n} \mathsf{guards}(l_i, t) & \text{if } t \text{ is a variable of sort } \alpha \\ \mathsf{true} & \text{otherwise} \end{cases}$$

A literal l guards a variable X according to the rules that we discussed in section 2.4.

$$\mathsf{guards}(l, X) = \begin{cases} p_T & \text{if } l \text{ is of the form } P(\ldots, X, \ldots), \\ p_F & \text{if } l \text{ is of the form } \neg P(\ldots, X, \ldots), \\ \mathsf{true} & \text{if } l \text{ is of the form } X \neq f(\ldots) \text{ or } f(\ldots) \neq X, \\ \mathsf{false} & \text{otherwise.} \end{cases}$$

If there is a satisfying assignment of the SAT-formula $\mathsf{mono}(\varphi, \alpha)$, then there is a consistent extension of the predicates of φ (a context) that makes φ monotonic in α, and vice versa. Monotonox uses MiniSat [Eén and Sörensson, 2003] to find out whether a satisfying assignment exists for each sort.

4 Translating Sorted to Unsorted Logic

To translate from a sorted problem to an unsorted problem, we use the principle that *monotonic sorts can simply be erased*, but non-monotonic sorts need to be encoded using, for example, a sort predicate. Thus our algorithm is as follows:

1. Analyse the formula to discover which sorts are monotonic.
2. For each non-monotonic sort, transform the formula by introducing a sort predicate or a sort function (according to the user's choice)—but do not erase the sort yet.
3. Erase all the sorts at once.

It makes no difference *how* we encode the non-monotonic sorts—by using predicates, functions or something else. We can in principle use sort predicates for some sorts and sort functions for others.

We justify the algorithm as follows: by adding sort predicates or functions for all the non-monotonic sorts, we have transformed the input formula into an equisatisfiable formula *which is also monotonic*.[4] Once we have this monotonic formula then erasing all the sorts preserves satisfiability (theorem 1).

An example Suppose we take the first axiom of our running example, $\forall M \in \mathit{monkey}.\, \mathsf{owns}(M, \mathsf{banana}_1(M))$. As discussed, we know that the sort *banana* is monotonic but the sort *monkey* is not. Thus we need to introduce a sort predicate or function for only the sort *monkey*. If we introduce a sort function—while still keeping the formula sorted—we obtain the formula $\forall M \in \mathit{monkey}.\, \mathsf{owns}(\mathsf{f}_{\mathit{monkey}}(M), \mathsf{banana}_1(\mathsf{f}_{\mathit{monkey}}(M))$.

Having done this, it is enough to erase the sorts from the formula (step 3 of the algorithm) and we obtain an unsorted formula which is equisatisfiable over each domain size to the original sorted formula, namely:

$$\forall M.\, \mathsf{owns}(\mathsf{f}_{\mathit{monkey}}(M), \mathsf{banana}_1(\mathsf{f}_{\mathit{monkey}}(M))$$

[4] In the case of sort predicates, our second calculus can infer monotonicity by false-extending the sort predicate; in the case of sort functions, our first calculus also can because no variable appears directly as the argument of an equality literal.

5 Translating Unsorted to Sorted Logic

The translation from unsorted to sorted formulae makes use of the same machinery, only in the reverse direction: given an unsorted problem ϕ, if we find a well-sorted problem ψ such that (1) erasing the sorts in ψ gives us back ϕ, and (2) all sorts in ψ are monotonic, then (theorem 1) ϕ and ψ are equisatisfiable.

The problem is finding the sorted problem ψ. We can use an existing algorithm [Claessen and Sörensson, 2003], that we call *sort unerasure* here, for this. Sort unerasure computes the *maximal typing* of an unsorted problem. It starts by creating unique sorts for all variable occurrences in the problem, and for all argument positions of predicate and function symbols, and for all results of function symbols. Then, it computes equivalence classes of sorts that should be equal to each other in order for the problem to be well-sorted, in the following way. Everywhere in the problem, whenever we apply a function symbol or predicate symbol P to a term t, we force the sort of the corresponding argument position of P to be in the same equivalence class as the result sort of t. Using a union/find algorithm, we get an algorithm that is close to linear time in complexity.

To sum up, the translation goes in three steps:

1. Compute candidate sorts for all symbols occurring in the problem (using sort unerasure), and create the corresponding sorted problem.

2. Use Monotonox to find out if all sorts in the resulting problem are monotonic. If they are, we are done.

3. If there exists any sort that cannot be shown monotonic, then give up. We simply return the unsorted problem as a sorted problem with one sort.

In practice, there is more we can do in step 3 than giving up. One has to constrain the sorted formula so that (1) the domains of all non-monotonic sorts have the same size, and (2) no monotonic sort's domain can be bigger than a non-monotonic sort's domain. A finite model finder can easily implement these constraints; when theorem-proving, one can enforce size constraints between sorts by adding to the problem an injective function from the smaller sort to the bigger sort.

6 Results

The TPTP library [Sutcliffe, 2009] has recently been extended with many-sorted (so-called TFF) problems. Unfortunately, only 26 of these problems have more than one sort.[5] They break down as follows: 11 have no non-ground positive equality, which means that they are trivially monotonic. Monotonox proves a further 5 monotonic. 4 are monotonic only because they have no finite models, a situation which we cannot detect but plan to in the future. 6 are truly not monotonic.

Translating from unsorted to many-sorted logic, we applied sort unerasure to all 13610 unsorted TPTP problems,[6] finding 6380 problems with more than one sort, to which we applied our monotonicity inference. The results are as follows.

	Monotonic sorts	Other sorts[7]	Affected problems[8]	Monotonic problems[9]
CNF problems (2598 problems, 19908 sorts)				
Calculus 1	12317	7591	2446	592
Calculus 2	12545	7363	2521	726
Full first-order problems (3782 problems, 91843 sorts)				
Calculus 1	85025	6818	3154	1034
Calculus 2	88645	3198	3715	1532

Running times None of the tests above took more than a few seconds. Monotonicity inference was not more expensive than the sort unerasure algorithm.

7 Conclusions and Future Work

We have introduced the concept of *monotonicity*, and applied it to the problem of translating between many-sorted and unsorted first-order logic. Detecting monotonicity of a sort is not decidable, but we have introduced two algorithms approximating the answer, one

[5] TFF adds both sorts and arithmetic to TPTP; the vast majority of the problems so far only test arithmetic, so only have one sort.

[6] Excluding the so-called SYN problems that just test syntax.

[7] Sorts that we couldn't infer monotonic (including sorts that are truly not monotonic).

[8] Problems where at least one sort was inferred monotonic.

[9] Problems where all sorts were inferred monotonic.

linear in the size of the problem, and one improved algorithm solving an NP-complete problem using a SAT-solver. Our results show that the improved algorithm detects many cases of monotonicity, and that the NP-completeness is not a problem in practice.

For future work, we plan to integrate our previous work on finite unsatisfiability detection [Claessen and Lillieström, 2011] with monotonicity detection—any sort which must have an infinite domain is monotonic. We expect this method to improve monotonicity detection for typical problems that have been translated from higher-order logics with recursive datatypes, such as lists. Moreover, we are working on generalising guards to arbitrary literals.

Finally, we plan to use the translation from unsorted to many-sorted logic to populate the typed section of the TPTP benchmark library.

4

Encoding Monomorphic and Polymorphic Types

This paper was published at TACAS 2013. This is an extended version with proofs.

Chapter 4

Encoding Monomorphic and Polymorphic Types

Jasmin Christian Blanchette Sascha Böhme
Andrei Popescu Nicholas Smallbone

Abstract

Most automatic theorem provers are restricted to untyped logics, and existing translations from typed logics are bulky or unsound. Recent research proposes monotonicity as a means to remove some clutter. Here we pursue this approach systematically, analysing formally a variety of encodings that further improve on efficiency while retaining soundness and completeness. We extend the approach to rank-1 polymorphism and present alternative schemes that lighten the translation of polymorphic symbols based on the novel notion of "cover". The new encodings are implemented, and partly proved correct, in Isabelle/HOL. Our evaluation finds them vastly superior to previous schemes.

1 Introduction

Specification languages, proof assistants, and other theorem proving applications typically rely on polymorphic types, but state-of-the-art automatic provers support only untyped or monomorphic logics. The existing sound and complete translation schemes for polymorphic types, whether they revolve around functions (tags) or predicates (guards), produce clutter that severely hampers the proof search, and lighter approaches based on type arguments are unsound [Meng and Paulson,

2008, Stickel, 1986]. As a result, application authors face a difficult choice between soundness and efficiency.

In Paper 3 we presented a pair of sound, complete, and efficient translations from monomorphic to untyped first-order logic with equality. The key insight is that *monotonic* types—types whose domain can be extended with new elements while preserving satisfiability—can be merged. The remaining types can be made monotonic by introducing protectors (tags or guards).

Example 1 (Monkey Village). Imagine a village of monkeys where each monkey owns at least two bananas:

$$\forall M : \mathit{monkey}.\ \mathsf{owns}(M, \mathsf{b}_1(M)) \wedge \mathsf{owns}(M, \mathsf{b}_2(M))$$
$$\forall M : \mathit{monkey}.\ \mathsf{b}_1(M) \not\approx \mathsf{b}_2(M)$$
$$\forall M_1, M_2 : \mathit{monkey}, B : \mathit{banana}.\ \mathsf{owns}(M_1, B) \wedge \mathsf{owns}(M_2, B) \rightarrow M_1 \approx M_2$$

The predicate $\mathsf{owns} : \mathit{monkey} \times \mathit{banana} \rightarrow \mathrm{o}$ associates monkeys with bananas, and the functions $\mathsf{b}_1, \mathsf{b}_2 : \mathit{monkey} \rightarrow \mathit{banana}$ witness the existence of each monkey's minimum supply of bananas. The axioms are satisfiable.

In the monkey village of Example 1, the type *banana* is monotonic, because any model with b bananas can be extended to a model with $b' > b$ bananas. In contrast, *monkey* is non-monotonic, because there can live at most $\lfloor b/2 \rfloor$ monkeys in a village with a finite supply of b bananas. Syntactically, the monotonicity of *banana* is inferrable from the absence of a positive equality $B \approx t$ or $t \approx B$, where B is a variable of type *banana* and t is arbitrary; such a literal would be needed to make the type non-monotonic.

The example can be encoded as follows, using the predicate $\mathsf{g}_{\mathit{monkey}}$ to guard against ill-typed instantiations of M, M_1, and M_2:

$$\exists M.\ \mathsf{g}_{\mathit{monkey}}(M)$$

$$\forall M.\ \mathsf{g}_{\mathit{monkey}}(M) \rightarrow \mathsf{owns}(M, \mathsf{b}_1(M)) \wedge \mathsf{owns}(M, \mathsf{b}_2(M))$$
$$\forall M.\ \mathsf{g}_{\mathit{monkey}}(M) \rightarrow \mathsf{b}_1(M) \not\approx \mathsf{b}_2(M)$$
$$\forall M_1, M_2, B.\ \mathsf{g}_{\mathit{monkey}}(M_1) \wedge \mathsf{g}_{\mathit{monkey}}(M_2) \wedge \mathsf{owns}(M_1, B) \wedge \mathsf{owns}(M_2, B) \rightarrow M_1 \approx M_2$$

(The first axiom witnesses the existence of a monkey.) Thanks to monotonicity, it is sound to omit all type information regarding bananas.

Monotonicity is not decidable, but it can often be inferred using suitable calculi. In this paper, we exploit this idea systematically, analysing

a variety of encodings based on monotonicity: some are minor adaptations of existing ones, while others are novel encodings that further improve on the size of the translated formulas.

In addition, we generalise the monotonicity approach to a rank-1 polymorphic logic, as embodied by the TPTP typed first-order form TFF1 [Blanchette and Paskevich, 2012]. Unfortunately, the presence of a single equality literal $X \approx t$ or $t \approx X$, where X is a polymorphic variable of type α, will lead the analysis to classify all types as possibly non-monotonic and force the use of protectors everywhere, as in the traditional encodings. A typical example is the list axiom $\forall X : \alpha,\ Xs : \mathtt{list}(\alpha).\ \mathsf{hd}(\mathsf{cons}(X, Xs)) \approx X$. We solve this issue through a novel scheme that reduces the clutter associated with non-monotonic types, based on the observation that protectors are required only when translating the particular formulas that prevent a type from being inferred monotonic. This contribution improves the monomorphic case as well: for the monkey village example, our scheme detects that the first two axioms are harmless and translates them without the g_{monkey} guards. In fact, by relying on a relaxed notion of monotonicity, we can soundly eliminate all type information in the monkey village problem.

Encoding types in an untyped logic is an old problem, and several solutions have been proposed in the literature. We first review four main traditional approaches (Section 3), which prepare the ground for the more advanced encodings presented in this paper. Next, we present known and novel monotonicity-based schemes that handle only ground types (Section 4); these are interesting in their own right and serve as stepping stones for the full-blown polymorphic encodings (Section 5). Besides the monotonicity-based encodings, we also present alternative schemes that aim at reducing the clutter associated with polymorphic symbols, based on the novel notion of "cover" (Section 6). Proofs of correctness accompany the descriptions of the new encodings. The proofs explicitly relate models of unencoded and encoded problems.

Figure 4.1 presents a brief overview of the main encodings. The traditional encodings are identified by single letters (e for full type erasure, a for type arguments, t for type tags, g for type guards). The non-traditional encodings append a suffix to the letter: ? (= monotonicity-based, lightweight), ?? (= monotonicity-based, featherweight), or @ (= cover-based). The decoration ~ identifies the monomorphic version of an encoding. Among the non-traditional schemes, $\widetilde{\mathsf{t}}$? and $\widetilde{\mathsf{g}}$? come from Paper 3; the other encodings are novel.

A formalisation [Blanchette and Popescu, 2012] of the results in the proof assistant Isabelle/HOL [Nipkow et al., 2002] is under way; it

Traditional	Monotonicity-based		Cover-based
Polymorphic	Monomorphic	Polymorphic	Polymorphic
e (§3.1)			
a (§3.2)		$\mathsf{a}^{\mathsf{ctor}}$ (§5.1)	
t (§3.3)	$\widetilde{\mathsf{t}}$?, $\widetilde{\mathsf{t}}$?? (§4.4)	t?, t?? (§5.4)	t@ (§6.1)
g (§3.4)	$\widetilde{\mathsf{g}}$?, $\widetilde{\mathsf{g}}$?? (§4.5)	g?, g?? (§5.5)	g@ (§6.2)

Figure 4.1: Main encodings

currently covers all the monomorphic encodings. The encodings have been implemented in Sledgehammer [Blanchette et al., 2011, Meng and Paulson, 2008], which provides a bridge between Isabelle/HOL and several automatic theorem provers (Section 7). They were evaluated with E, iProver, SPASS, Vampire, and Z3 on a vast benchmark suite consisting of proof goals from existing Isabelle formalisations (Section 8). Our comparisons include the traditional encodings as well as the provers' native support for monomorphic types where it is available. Related work is considered at the end (Section 9).

2 Background: Logics

This paper involves three versions of classical first-order logic with equality: polymorphic, monomorphic, and untyped. They correspond to the TPTP syntaxes TFF1, TFF0, and FOF, respectively, excluding interpreted arithmetic.

2.1 Polymorphic First-Order Logic

The source logic is a rank-1 polymorphic logic as provided by TFF1 [Blanchette and Paskevich, 2012].

Definition 1 (Syntax). Let $\mathcal{A}$ be a countable set of *type variables* with typical element α, and let $\mathcal{V}$ be a countable set of *term variables* with typical element X. A *polymorphic signature* is a triple $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$, where $\mathcal{K}$ is a finite set of n-ary type constructors k with their arities, $\mathcal{F}$ is a finite set of function symbol declarations, and $\mathcal{P}$ is a finite set of predicate symbol declarations. The types, declarations, terms, and formulas are defined below. Symbols may not be overloaded. A *problem* over Σ is a finite set of closed formulas over Σ.

Types:

σ	$::=$	$k(\bar{\sigma})$	constructor type
	$\mid$	α	type variable

Declarations:

$$f : \forall\bar{\alpha}.\ \sigma_1 \times \cdots \times \sigma_n \to \sigma \in \mathcal{F}$$
$$p : \forall\bar{\alpha}.\ \sigma_1 \times \cdots \times \sigma_n \to o \in \mathcal{P}$$

The type variables $\bar{\alpha}$ in a declaration must be distinct and comprise all type variables found in $\sigma_1, \ldots, \sigma_n$ and σ. Type variables α_i that do not occur in $\sigma_1, \ldots, \sigma_n$ or σ are allowed. The symbols $\times$, $\to$, and o are not type constructors but syntax. Both kinds of declaration are instances of the general syntax $s : \forall\bar{\alpha}.\ \bar{\sigma} \to \sigma$, where $s \in \mathcal{F} \uplus \mathcal{P}$ and σ is either a type or o. An application of s requires $|\bar{\alpha}|$ type arguments in angle brackets and $|\bar{\sigma}|$ term arguments in parentheses. In examples, we usually omit type arguments that are irrelevant or clear from the context.

Terms:

t	$::=$	$f\langle\bar{\sigma}\rangle(\bar{t})$	function term
	$\mid$	X	term variable

Formulas:

φ	$::=$	$p\langle\bar{\sigma}\rangle(\bar{t}) \mid \neg\, p\langle\bar{\sigma}\rangle(\bar{t})$	predicate literal
	$\mid$	$t_1 \approx t_2 \mid t_1 \not\approx t_2$	equality literal
	$\mid$	$\varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$	binary connective
	$\mid$	$\forall X : \sigma.\ \varphi \mid \exists X : \sigma.\ \varphi$	term quantification
	$\mid$	$\forall\alpha.\ \varphi$	type quantification

Though it is not captured syntactically, a type quantifier may not appear underneath a term quantifier. When defining translations or reasoning about formulas, we assume they are expressed in negation normal form (NNF), with negation applied to equality or predicate atoms. In other contexts, we freely nest quantifiers and connectives and use implication $\to$. We assume that all type quantification is universal—type skolemisation will remove existential type quantifiers [Blanchette and Paskevich, 2012, §5.2]. Finally, we adopt the convention that each type or term variable is bound only once in a formula.

The typing rules and semantics of the logic are modelled after those of TFF1 [Blanchette and Paskevich, 2012, §3]. Briefly, the type arguments completely determine the types of the term arguments and, for functions, of the result. Polymorphic symbols are interpreted as families of

functions or predicates indexed by ground types. All types are inhabited (non-empty).

Definition 2 (Typing Rules)**.** Let Γ be a *type context*, a function that maps every variable to its type. A judgement $\Gamma \vdash t : \sigma$ expresses that the term t is *well-typed* and has type σ in context Γ. A judgement $\Gamma \vdash \varphi : o$ expresses that the formula φ is *well-typed* in Γ. The *typing rules* of polymorphic first-order logic are given below:

$$\frac{}{\Gamma \vdash X : \Gamma(X)} \qquad \frac{f : \forall \bar{\alpha}.\ \bar{\sigma} \to \sigma \qquad \Gamma \vdash t_j : \sigma_j\,\rho \ \text{ for all } j}{\Gamma \vdash f\langle \bar{\alpha}\rho\rangle(\bar{t}\,) : \sigma\rho}$$

$$\frac{p : \forall \bar{\alpha}.\ \bar{\sigma} \to o \qquad \Gamma \vdash t_j : \sigma_j\,\rho \ \text{ for all } j}{\Gamma \vdash p\langle \bar{\alpha}\rho\rangle(\bar{t}\,) : o}$$

$$\frac{p : \forall \bar{\alpha}.\ \bar{\sigma} \to o \qquad \Gamma \vdash t_j : \sigma_j\,\rho \ \text{ for all } j}{\Gamma \vdash \neg\, p\langle \bar{\alpha}\rho\rangle(\bar{t}\,) : o}$$

$$\frac{\Gamma \vdash t_1 : \sigma \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1 \approx t_2 : o} \qquad \frac{\Gamma \vdash t_1 : \sigma \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1 \not\approx t_2 : o}$$

$$\frac{\Gamma \vdash \varphi_1 : o \quad \Gamma \vdash \varphi_2 : o}{\Gamma \vdash \varphi_1 \wedge \varphi_2 : o} \qquad \frac{\Gamma \vdash \varphi_1 : o \quad \Gamma \vdash \varphi_2 : o}{\Gamma \vdash \varphi_1 \vee \varphi_2 : o}$$

$$\frac{\Gamma[X \mapsto \sigma] \vdash \varphi : o}{\Gamma \vdash \forall X : \sigma.\ \varphi : o} \qquad \frac{\Gamma[X \mapsto \sigma] \vdash \varphi : o}{\Gamma \vdash \exists X : \sigma.\ \varphi : o} \qquad \frac{\Gamma \vdash \varphi : o}{\Gamma \vdash \forall \alpha.\ \varphi : o}$$

Superscripts attach type annotations to terms; for example, t^σ indicates that t has type σ. These annotations are a notational device and not part of the syntax.

Definition 3 (Semantics)**.** Let $\mathcal{D}$ be a fixed non-empty collection of non-empty sets (the *domains*), and let $\mathcal{U} = \bigcup \mathcal{D}$ (the *universe*). Let $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ be a polymorphic signature. A *structure* $\mathcal{S}$ for Σ is a triple of families:

- $(k^{\mathcal{S}})_{k \in \mathcal{K}} : \mathcal{D}^n \to \mathcal{D}$;
- $(f^{\mathcal{S}})_{f\,:\,\forall \bar{\alpha}.\ \bar{\sigma}\to\sigma \,\in\, \mathcal{F}} : \mathcal{D}^{|\bar{\alpha}|} \times \mathcal{U}^{|\bar{\sigma}|} \to \mathcal{U}$;
- $(p^{\mathcal{S}})_{p\,:\,\forall \bar{\alpha}.\ \bar{\sigma}\to o \,\in\, \mathcal{P}} \subseteq \mathcal{D}^{|\bar{\alpha}|} \times \mathcal{U}^{|\bar{\sigma}|}$.

Given a *type variable valuation* $\theta : \mathcal{A} \to \mathcal{D}$, the *interpretation* of types $[\![\ ]\!]^{\mathcal{S}}_{\theta}$ is defined by the equations

$$[\![k(\bar{\sigma})]\!]^{\mathcal{S}}_{\theta} = k^{\mathcal{S}}([\![\bar{\sigma}]\!]^{\mathcal{S}}_{\theta}) \qquad [\![\alpha]\!]^{\mathcal{S}}_{\theta} = \theta(\alpha)$$

The $p^{\mathcal{S}}$ component of a structure is required to map any tuple of $|\bar{\alpha}|$ domains D_i and $|\bar{\sigma}|$ universe elements $a_j \in [\![\sigma_j]\!]^{\mathcal{S}}_{\theta}$ to an element of $[\![\sigma]\!]^{\mathcal{S}}_{\theta}$, where θ maps each α_i to D_i.

Given a type variable valuation θ and a *term variable valuation* $\xi : \mathcal{V} \to \mathcal{U}$, the *interpretation* of terms and formulas by the structure $\mathcal{S}$ is as follows:

$$\begin{aligned}
[\![f\langle\bar{\sigma}\rangle(\bar{t})]\!]^{\mathcal{S}}_{\theta,\,\xi} &= f^{\mathcal{S}}([\![\bar{\sigma}]\!]^{\mathcal{S}}_{\theta},\, [\![\bar{t}\,]\!]^{\mathcal{S}}_{\theta,\,\xi}) \\
[\![p\langle\bar{\sigma}\rangle(\bar{t})]\!]^{\mathcal{S}}_{\theta,\,\xi} &= p^{\mathcal{S}}([\![\bar{\sigma}]\!]^{\mathcal{S}}_{\theta},\, [\![\bar{t}\,]\!]^{\mathcal{S}}_{\theta,\,\xi}) \\
[\![\neg\, p\langle\bar{\sigma}\rangle(\bar{t})]\!]^{\mathcal{S}}_{\theta,\,\xi} &= \neg\, p^{\mathcal{S}}([\![\bar{\sigma}]\!]^{\mathcal{S}}_{\theta},\, [\![\bar{t}\,]\!]^{\mathcal{S}}_{\theta,\,\xi}) \\
[\![X]\!]^{\mathcal{S}}_{\theta,\,\xi} &= \xi(X) \\
[\![t_1 \approx t_2]\!]^{\mathcal{S}}_{\theta,\,\xi} &= ([\![t_1]\!]^{\mathcal{S}}_{\theta,\,\xi} = [\![t_2]\!]^{\mathcal{S}}_{\theta,\,\xi}) \\
[\![t_1 \not\approx t_2]\!]^{\mathcal{S}}_{\theta,\,\xi} &= ([\![t_1]\!]^{\mathcal{S}}_{\theta,\,\xi} \neq [\![t_2]\!]^{\mathcal{S}}_{\theta,\,\xi}) \\
[\![\varphi_1 \wedge \varphi_2]\!]^{\mathcal{S}}_{\theta,\,\xi} &= [\![\varphi_1]\!]^{\mathcal{S}}_{\theta,\,\xi} \wedge [\![\varphi_2]\!]^{\mathcal{S}}_{\theta,\,\xi} \\
[\![\varphi_1 \vee \varphi_2]\!]^{\mathcal{S}}_{\theta,\,\xi} &= [\![\varphi_1]\!]^{\mathcal{S}}_{\theta,\,\xi} \vee [\![\varphi_2]\!]^{\mathcal{S}}_{\theta,\,\xi} \\
[\![\forall X : \sigma.\ \varphi]\!]^{\mathcal{S}}_{\theta,\,\xi} &= \forall a \in [\![\sigma]\!]^{\mathcal{S}}_{\theta}.\ [\![\varphi]\!]^{\mathcal{S}}_{\theta,\,\xi[X \mapsto a]} \\
[\![\exists X : \sigma.\ \varphi]\!]^{\mathcal{S}}_{\theta,\,\xi} &= \exists a \in [\![\sigma]\!]^{\mathcal{S}}_{\theta}.\ [\![\varphi]\!]^{\mathcal{S}}_{\theta,\,\xi[X \mapsto a]} \\
[\![\forall \alpha.\ \varphi]\!]^{\mathcal{S}}_{\theta,\,\xi} &= \forall D \in \mathcal{D}.\ [\![\varphi]\!]^{\mathcal{S}}_{\theta[\alpha \mapsto D],\,\xi}
\end{aligned}$$

We omit irrelevant subscripts to $[\![\ ]\!]$, writing $[\![\sigma]\!]^{\mathcal{S}}$ if σ is ground and $[\![\varphi]\!]^{\mathcal{S}}$ if φ is closed.

A structure $\mathcal{M}$ is a *model* of a problem Φ if $[\![\varphi]\!]^{\mathcal{M}}$ is true for every $\varphi \in \Phi$. A problem that has a model is *satisfiable*.

Example 2 (Algebraic Lists). The following axioms induce a minimalistic first-order theory of algebraic lists that will serve as our main running example:

$$\begin{aligned}
&\forall \alpha.\ \forall X : \alpha,\ Xs : \mathtt{list}(\alpha).\ \mathsf{nil} \not\approx \mathsf{cons}(X, Xs) \\
&\forall \alpha.\ \forall Xs : \mathtt{list}(\alpha). \\
&\quad Xs \approx \mathsf{nil} \vee (\exists Y : \alpha,\ Ys : \mathtt{list}(\alpha).\ Xs \approx \mathsf{cons}(Y, Ys)) \\
&\forall \alpha.\ \forall X : \alpha,\ Xs : \mathtt{list}(\alpha). \\
&\quad \mathsf{hd}(\mathsf{cons}(X, Xs)) \approx X \wedge \mathsf{tl}(\mathsf{cons}(X, Xs)) \approx Xs
\end{aligned}$$

We conjecture that cons is injective. The conjecture's negation can be expressed employing an unknown but fixed Skolem type b:

$$\exists X, Y : \mathsf{b},\ Xs, Ys : \mathtt{list}(\mathsf{b}).\\ \quad \mathsf{cons}(X, Xs) \approx \mathsf{cons}(Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)$$

Because the hd and tl equations force injectivity of cons in both arguments, the problem is unsatisfiable: the unnegated conjecture is a consequence of the axioms.

Central to this paper are the notions of soundness and completeness of a translation function between problems.

Definition 4 (Correctness)**.** Assume a function that translates problems Φ over Σ to problems Φ' over Σ'. The function is *sound* if satisfiability of Φ implies satisfiability of Φ'; it is *complete* if satisfiability of Φ' implies satisfiability of Φ; it is *correct* if it is both sound and complete (i.e. Φ and Φ' are equisatisfiable).

2.2 Monomorphic First-Order Logic

Monomorphic first-order logic constitutes a special case of polymorphic first-order logic. Type constructors are nullary, symbol declarations have the form $\mathsf{s} : \bar{\sigma} \to \sigma$, and formulas contain no type variables.

Example 3. A monomorphised version of the algebraic list problem of Example 2, with α instantiated by b, follows:

$$\forall X : \mathsf{b},\ Xs : \mathtt{list_b}.\\ \quad \mathsf{nil_b} \not\approx \mathsf{cons_b}(X, Xs)\\ \forall Xs : \mathtt{list_b}.\\ \quad Xs \approx \mathsf{nil_b} \vee (\exists Y : \mathsf{b},\ Ys : \mathtt{list_b}.\ Xs \approx \mathsf{cons_b}(Y, Ys))\\ \forall X : \mathsf{b},\ Xs : \mathtt{list_b}.\\ \quad \mathsf{hd_b}(\mathsf{cons_b}(X, Xs)) \approx X \wedge \mathsf{tl_b}(\mathsf{cons_b}(X, Xs)) \approx Xs\\ \exists X, Y : \mathsf{b},\ Xs, Ys : \mathtt{list_b}.\\ \quad \mathsf{cons_b}(X, Xs) \approx \mathsf{cons_b}(Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)$$

2.3 Untyped First-Order Logic

Our target logic is an untyped logic coinciding with the TPTP first-order form FOF [Sutcliffe, 2009]. This is the logic traditionally implemented in automatic theorem provers. An *untyped signature* is a pair $\Sigma = (\mathcal{F}, \mathcal{P})$,

where $\mathcal{F}$ and $\mathcal{P}$ associate symbols with their arities (indicated by superscripts). The untyped syntax is identical to that of the monomorphic logic, except that quantification is written $\forall X.\ \varphi$ and $\exists X.\ \varphi$.

3 Traditional Type Encodings

There are four main traditional approaches to encoding polymorphic types: full type erasure, type arguments, type tags, and type guards [Enderton, 2001, Meng and Paulson, 2008, Stickel, 1986, Wick and McCune, 1989].

3.1 Full Type Erasure

The easiest way to translate a typed problem into an untyped logic is to erase all its type information, which means omitting all type arguments, type quantifiers, and types in term quantifiers. We call this encoding e.

Definition 5 (Full Erasure e). The *full type erasure* encoding e translates a polymorphic problem over Σ into an untyped problem over Σ', where the symbols in Σ' have the same term arities as in Σ (but without type arguments). It is defined on terms and formulas by the following structurally recursive function $[\![\ ]\!]_{\mathsf{e}}$:

$$
\begin{aligned}
[\![\mathsf{f}\langle\bar{\sigma}\rangle(\bar{t}\,)]\!]_{\mathsf{e}} &= \mathsf{f}([\![\bar{t}\,]\!]_{\mathsf{e}}) \\
[\![\mathsf{p}\langle\bar{\sigma}\rangle(\bar{t}\,)]\!]_{\mathsf{e}} &= \mathsf{p}([\![\bar{t}\,]\!]_{\mathsf{e}}) \qquad & [\![\forall X : \sigma.\ \varphi]\!]_{\mathsf{e}} &= \forall X.\ [\![\varphi]\!]_{\mathsf{e}} \\
[\![\neg\, \mathsf{p}\langle\bar{\sigma}\rangle(\bar{t}\,)]\!]_{\mathsf{e}} &= \neg\, \mathsf{p}([\![\bar{t}\,]\!]_{\mathsf{e}}) \qquad & [\![\exists X : \sigma.\ \varphi]\!]_{\mathsf{e}} &= \exists X.\ [\![\varphi]\!]_{\mathsf{e}} \\
[\![\forall \alpha.\ \varphi]\!]_{\mathsf{e}} &= [\![\varphi]\!]_{\mathsf{e}}
\end{aligned}
$$

Here and elsewhere, we omit the trivial cases where the function is simply applied to its subterms or subformulas, as in $[\![\varphi_1 \wedge \varphi_2]\!]_{\mathsf{e}} = [\![\varphi_1]\!]_{\mathsf{e}} \wedge [\![\varphi_2]\!]_{\mathsf{e}}$. The translation of a problem Φ is the union of the translations its formulas: $[\![\Phi]\!]_{\mathsf{e}} = \bigcup_{\varphi \in \Phi} [\![\varphi]\!]_{\mathsf{e}}$.

By way of composition, the e encoding lies at the heart of all the encodings presented in this paper. Given n encodings $x_1, \ldots, x_n$ (where x_n is usually e), we write $[\![\ ]\!]_{x_1;\ldots;x_n}$ for the composition $[\![\ ]\!]_{x_n} \circ \cdots \circ [\![\ ]\!]_{x_1}$.

Example 4. Encoded using e, the monkey village axioms of Example 1 become

$$
\begin{aligned}
&\forall M.\ \mathsf{owns}(M, \mathsf{b}_1(M)) \wedge \mathsf{owns}(M, \mathsf{b}_2(M)) \\
&\forall M.\ \mathsf{b}_1(M) \not\approx \mathsf{b}_2(M) \\
&\forall M_1, M_2, B.\ \mathsf{owns}(M_1, B) \wedge \mathsf{owns}(M_2, B) \rightarrow M_1 \approx M_2
\end{aligned}
$$

Like the original axioms, the encoded axioms are satisfiable: the requirement that each monkey possesses two bananas of its own can be met by taking an infinite domain (since $2\kappa = \kappa$ for any infinite cardinal κ).

Full type erasure is unsound in the presence of equality because equality can be used to encode cardinality constraints on domains. For example, $\forall U : \mathsf{unit}.\ U \approx \mathsf{unity}$ forces the domain of unit to have only one element. Its erasure, $\forall U.\ U \approx \mathsf{unity}$, effectively restricts *all* types to one element; from any disequality $t \not\approx u$ or any pair of clauses $p(\bar{t})$ and $\neg p(\bar{u})$, we can derive a contradiction. An expedient proposed by Meng and Paulson [2008, §2.8] and implemented in Sledgehammer is to filter out all axioms of the form $\forall X : \sigma.\ X \approx a_1 \vee \cdots \vee X \approx a_n$, but this makes the translation incomplete and generally does not suffice to prevent unsound cardinality reasoning.

An additional issue with full type erasure is that it confuses distinct monomorphic instances of polymorphic symbols. The formula $q\langle a\rangle(f\langle a\rangle) \wedge \neg\ q\langle b\rangle(f\langle b\rangle)$ is satisfiable, but its type erasure $q(f) \wedge \neg\ q(f)$ is unsatisfiable. A less contrived example is $\mathsf{N} \not\approx 0 \to \mathsf{N} > 0$, which we would expect to hold for the natural number versions of 0 and $>$ but not for integers or real numbers.

Nonetheless, full type erasure is complete, and this property will be useful later.

Theorem 2 (Completeness of e). *Full type erasure is complete.*

Proof. From a model $\mathcal{M}$ of $[\![\Phi]\!]_e$, we construct a structure for Φ by taking the same domain for all types and interpreting all instances of each polymorphic symbol in the same way as $\mathcal{M}$. This construction clearly yields a model of Φ. □

3.2 Type Arguments

A way to prevent the confusion arising with full type erasure is to encode types as terms in the untyped logic: type variables α become term variables A, and n-ary type constructors κ become n-ary function symbols κ. A polymorphic symbol with m type arguments is passed m additional term arguments. The example given in the previous subsection is translated to $q(a, f(a)) \wedge \neg\ q(b, f(b))$, and a fully polymorphic instance $f\langle\alpha\rangle$ would be encoded as $f(A)$. We call this encoding a.

Definition 6 (Term Encoding of Types). Let $\mathcal{K}$ be a finite set of n-ary type constructors, and let $\vartheta \notin \mathcal{K}$ be a distinguished nullary type constructor.

For each type variable α, let $\mathcal{V}(\alpha)$ be a fresh term variable. The *term encoding* of a polymorphic type over $\mathcal{K}$ is a term over $(\{\vartheta\}, \mathcal{K})$, where $\mathsf{k} : \vartheta^n \to \vartheta$ for each n-ary type constructor k. The encoding is specified by the following equations:

$$\langle\!\langle \mathsf{k}(\bar{\sigma}) \rangle\!\rangle = \mathsf{k}(\langle\!\langle \bar{\sigma} \rangle\!\rangle) \qquad\qquad \langle\!\langle \alpha \rangle\!\rangle = \mathcal{V}(\alpha)$$

Definition 7 (Traditional Arguments a)**.** The *traditional type arguments* encoding a translates a polymorphic problem over $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \mathcal{K},\ \mathcal{P}')$, where the symbols in $\mathcal{F}'$, $\mathcal{P}'$ are the same as those in $\mathcal{F}$, $\mathcal{P}$. For each symbol $\mathsf{s} : \forall\bar{\alpha}.\ \bar{\sigma} \to \sigma \in \mathcal{F} \uplus \mathcal{P}$, the arity of s in Σ' is $|\bar{\alpha}| + |\bar{\sigma}|$. The encoding is defined as $[\![\ ]\!]_{\mathsf{a};\mathsf{e}}$, where

$$\begin{aligned}
[\![\mathsf{f}\langle\bar{\sigma}\rangle(\bar{\mathsf{t}})]\!]_{\mathsf{a}} &= \mathsf{f}\langle\bar{\sigma}\rangle(\langle\!\langle\bar{\sigma}\rangle\!\rangle,\ [\![\bar{\mathsf{t}}]\!]_{\mathsf{a}}) \\
[\![\mathsf{p}\langle\bar{\sigma}\rangle(\bar{\mathsf{t}})]\!]_{\mathsf{a}} &= \mathsf{p}\langle\bar{\sigma}\rangle(\langle\!\langle\bar{\sigma}\rangle\!\rangle,\ [\![\bar{\mathsf{t}}]\!]_{\mathsf{a}}) \qquad [\![\forall\alpha.\ \varphi]\!]_{\mathsf{a}} = \forall\alpha.\ \forall\langle\!\langle\alpha\rangle\!\rangle : \vartheta.\ [\![\varphi]\!]_{\mathsf{a}} \\
[\![\neg\ \mathsf{p}\langle\bar{\sigma}\rangle(\bar{\mathsf{t}})]\!]_{\mathsf{a}} &= \neg\ \mathsf{p}\langle\bar{\sigma}\rangle(\langle\!\langle\bar{\sigma}\rangle\!\rangle,\ [\![\bar{\mathsf{t}}]\!]_{\mathsf{a}})
\end{aligned}$$

(Again, we omit the trivial cases, e.g. $[\![\forall X : \sigma.\ \varphi]\!]_{\mathsf{a}} = \forall X : \sigma.\ [\![\varphi]\!]_{\mathsf{a}}$.)

Example 5. The a encoding translates the algebraic list problem of Example 2 into the following untyped problem:

$$\begin{aligned}
&\forall A,\ X,\ Xs. \\
&\quad \mathsf{nil}(A) \not\approx \mathsf{cons}(A,\ X,\ Xs) \\
&\forall A,\ Xs. \\
&\quad Xs \approx \mathsf{nil}(A) \vee (\exists Y,\ Ys.\ Xs \approx \mathsf{cons}(A,\ Y,\ Ys)) \\
&\forall A,\ X,\ Xs. \\
&\quad \mathsf{hd}(A,\ \mathsf{cons}(A,\ X,\ Xs)) \approx X \wedge \mathsf{tl}(A,\ \mathsf{cons}(A,\ X,\ Xs)) \approx Xs \\
&\exists X,\ Y,\ Xs,\ Ys. \\
&\quad \mathsf{cons}(\mathsf{b},\ X,\ Xs) \approx \mathsf{cons}(\mathsf{b},\ Y,\ Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)
\end{aligned}$$

The a encoding coincides with e for monomorphic problems and is unsound. Nonetheless, it forms the basis of all our sound polymorphic encodings in a slightly generalised version, called a^x below. First, let us fix a distinguished type ϑ (for encoded types) and two distinguished symbols $\mathsf{t} : \forall\alpha.\ \alpha \to \alpha$ (for tags) and $\mathsf{g} : \forall\alpha.\ \alpha \to \mathsf{o}$ (for guards).

Definition 8 (Type Argument Filter)**.** Given a signature $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$, a *type argument filter* x maps any $\mathsf{s} : \forall\alpha_1, \ldots, \alpha_m.\ \bar{\sigma} \to \sigma$ to a subset $x_{\mathsf{s}} = \{i_1, \ldots, i_{m'}\} \subseteq \{1, \ldots, m\}$ of its type argument indices. Given a list

$\bar{z}$ of length m, $x_s(\bar{z})$ denotes the sublist $z_{i_1}, \ldots, z_{i_{m'}}$, where $i_1 < \cdots < i_{m'}$. Filters are implicitly extended to $\{1\}$ for the distinguished symbols $t, g \notin \mathcal{F} \uplus \mathcal{P}$.

Definition 9 (Generic Arguments a^x)**.** Given a type argument filter x, the *generic type arguments* encoding a^x translates a polymorphic problem over $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \mathcal{K}, \mathcal{P}')$, where the symbols in $\mathcal{F}'$, $\mathcal{P}'$ are the same as those in $\mathcal{F}$, $\mathcal{P}$. For each symbol $s : \forall\bar{\alpha}.\ \bar{\sigma} \to \sigma \in \mathcal{F} \uplus \mathcal{P}$, the arity of s in Σ' is $|x_s| + |\bar{\sigma}|$. The encoding is defined as $[\![\ ]\!]_{a^x;e}$, where the non-trivial cases are

$$\begin{aligned}
[\![f\langle\bar{\sigma}\rangle(\bar{t}\,)]\!]_{a^x} &= f\langle\bar{\sigma}\rangle(\langle\!\langle x_f(\bar{\sigma})\rangle\!\rangle, [\![\bar{t}\,]\!]_{a^x}) \\
[\![p\langle\bar{\sigma}\rangle(\bar{t}\,)]\!]_{a^x} &= p\langle\bar{\sigma}\rangle(\langle\!\langle x_p(\bar{\sigma})\rangle\!\rangle, [\![\bar{t}\,]\!]_{a^x}) \\
[\![\neg\, p\langle\bar{\sigma}\rangle(\bar{t}\,)]\!]_{a^x} &= \neg\, p\langle\bar{\sigma}\rangle(\langle\!\langle x_p(\bar{\sigma})\rangle\!\rangle, [\![\bar{t}\,]\!]_{a^x}) \\
[\![\forall\alpha.\ \varphi]\!]_{a^x} &= \forall\alpha.\ \forall\langle\!\langle\alpha\rangle\!\rangle : \vartheta.\ [\![\varphi]\!]_{a^x}
\end{aligned}$$

The e and a encodings correspond to the special cases of a^x where x returns none or all of its arguments, respectively.

Theorem 3 (Completeness of a^x)**.** *The type arguments encoding* a^x *is complete.*

Proof. By Theorem 2, it suffices to construct a model $\mathcal{M}'$ of Φ from a model $\mathcal{M}$ of $[\![\Phi]\!]_{a^x}$. We let $\mathcal{M}'$ interpret each type constructor in $\mathcal{K}$ in the same way as $\mathcal{M}$. For each symbol $s \in \mathcal{F} \uplus \mathcal{P}$, the entry for $s\langle\bar{\sigma}\rangle(\bar{a})$ in $\mathcal{M}'$ is given by the interpretation of $s\langle\bar{\sigma}\rangle(\langle\!\langle x_s(\bar{\sigma}\,)\rangle\!\rangle, \bar{a})$ in $\mathcal{M}$. This construction obviously yields a model of Φ. □

3.3 Type Tags

An intuitive approach to encode type information soundly (as well as completely) is to wrap each term and subterm with its type using type tags. For polymorphic type systems, this scheme relies on a distinguished binary function $t(\langle\!\langle\sigma\rangle\!\rangle, t)$ that "annotates" each term t with its type σ. The tags make most type arguments superfluous. We call this scheme t, after the tag function of the same name. It is defined as a two-stage process: the first stage adds tags $t\langle\sigma\rangle(t)$ while preserving the polymorphism; the second stage encodes t's type argument as well as any phantom type arguments.

Definition 10 (Phantom Type Argument)**.** Let $\mathsf{s} : \forall\alpha_1, \ldots, \alpha_m.\ \bar{\sigma} \to \sigma \in \mathcal{F} \uplus \mathcal{P}$. The ith type argument is a *phantom* if α_i does not occur in $\bar{\sigma}$ or σ. Given a list $\bar{z} \equiv z_1, \ldots, z_m$, $\mathsf{phan}_\mathsf{s}(\bar{z})$ denotes the sublist $z_{i_1}, \ldots, z_{i_{m'}}$ corresponding to the phantom type arguments.

Definition 11 (Traditional Tags t)**.** The *traditional type tags* encoding t translates a polymorphic problem over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \mathcal{K} \uplus \{\mathsf{t}^2\}, \mathcal{P}')$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for $\mathsf{a}^{\mathsf{phan}}$ (i.e. a^x with $x = \mathsf{phan}$). It is defined as $[\![\ ]\!]_{\mathsf{t};\mathsf{a}^{\mathsf{phan}};\mathsf{e}}$, i.e. the composition of $[\![\ ]\!]_\mathsf{t}$, $[\![\ ]\!]_{\mathsf{a}^{\mathsf{phan}}}$, and $[\![\ ]\!]_\mathsf{e}$, where

$$[\![\mathsf{f}\langle\sigma\rangle(\bar{\mathsf{t}})]\!]_\mathsf{t} = \lfloor \mathsf{f}\langle\sigma\rangle([\![\bar{\mathsf{t}}]\!]_\mathsf{t})\rfloor \quad [\![\mathsf{X}]\!]_\mathsf{t} = \lfloor \mathsf{X}\rfloor \quad \text{with } \lfloor \mathsf{t}^\sigma\rfloor = \mathsf{t}\langle\sigma\rangle(\mathsf{t})$$

Example 6. The t encoding translates the algebraic list problem of Example 2 into the following equisatisfiable untyped problem:

$$\begin{aligned}
&\forall A, X, Xs.\\
&\quad \mathsf{t}(\mathsf{list}(A), \mathsf{nil}) \not\approx \mathsf{t}(\mathsf{list}(A), \mathsf{cons}(\mathsf{t}(A, X), \mathsf{t}(\mathsf{list}(A), Xs)))\\
&\forall A, Xs.\ \mathsf{t}(\mathsf{list}(A), Xs) \approx \mathsf{t}(\mathsf{list}(A), \mathsf{nil}) \vee\\
&\quad (\exists Y, Ys.\ \mathsf{t}(\mathsf{list}(A), Xs) \approx \mathsf{t}(\mathsf{list}(A), \mathsf{cons}(\mathsf{t}(A, Y), \mathsf{t}(\mathsf{list}(A), Ys))))\\
&\forall A, X, Xs.\\
&\quad \mathsf{t}(A, \mathsf{hd}(\mathsf{t}(\mathsf{list}(A), \mathsf{cons}(\mathsf{t}(A, X), \mathsf{t}(\mathsf{list}(A), Xs))))) \approx \mathsf{t}(A, X) \wedge\\
&\quad \mathsf{t}(\mathsf{list}(A), \mathsf{tl}(\mathsf{t}(\mathsf{list}(A), \mathsf{cons}(\mathsf{t}(A, X), \mathsf{t}(\mathsf{list}(A), Xs))))) \approx \mathsf{t}(\mathsf{list}(A), Xs)\\
&\exists X, Y, Xs, Ys.\\
&\quad \mathsf{t}(\mathsf{list}(\mathsf{b}), \mathsf{cons}(\mathsf{t}(\mathsf{b}, X), \mathsf{t}(\mathsf{list}(\mathsf{b}), Xs))) \approx\\
&\qquad \mathsf{t}(\mathsf{list}(\mathsf{b}), \mathsf{cons}(\mathsf{t}(\mathsf{b}, Y), \mathsf{t}(\mathsf{list}(\mathsf{b}), Ys))) \wedge\\
&\quad (\mathsf{t}(\mathsf{b}, X) \not\approx \mathsf{t}(\mathsf{b}, Y) \vee \mathsf{t}(\mathsf{list}(\mathsf{b}), Xs) \not\approx \mathsf{t}(\mathsf{list}(\mathsf{b}), Ys))
\end{aligned}$$

Since there are no phantoms in this example, all type information is carried by the t function's first argument.

3.4 Type Guards

Type tags heavily burden the terms. An alternative is to introduce type guards, which are predicates that restrict the range of variables. They take the form of a distinguished predicate $\mathsf{g}(\langle\!\langle\sigma\rangle\!\rangle, \mathsf{t})$ that checks whether t has type σ.

With the type tags encoding, only phantom type arguments needed to be encoded; here, we must encode any type arguments that cannot be read off the types of the term arguments. Thus, the type argument

is encoded for $\mathsf{nil}\langle\alpha\rangle$ (which has no term arguments) but omitted for $\mathsf{cons}\langle\alpha\rangle(X, Xs)$, $\mathsf{hd}\langle\alpha\rangle(Xs)$, and $\mathsf{tl}\langle\alpha\rangle(Xs)$.

Definition 12 (Inferable Type Argument)**.** Let $s : \forall\alpha_1, \ldots, \alpha_m.\ \bar{\sigma} \to \sigma \in \mathcal{F} \uplus \mathcal{P}$. A type argument is *inferrable* if it occurs in some of the term arguments' types. Given a list $\bar{z} \equiv z_1, \ldots, z_m$, $\mathsf{inf}_s(\bar{z})$ denotes the sublist $z_{i_1}, \ldots, z_{i_{m'}}$ corresponding to the inferrable type arguments, and $\mathsf{ninf}_s(\bar{z})$ denotes the sublist for non-inferrable type arguments.

Definition 13 (Traditional Guards g)**.** The *traditional type guards* encoding g translates a polymorphic problem over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \mathcal{K}, \mathcal{P}' \uplus \{g^2\})$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for a^{ninf}. It is defined as $[\![\ ]\!]_{g;a^{\mathsf{ninf}};e}$, where

$$[\![\forall X : \sigma.\ \varphi]\!]_g = \forall X : \sigma.\ g\langle\sigma\rangle(X) \to [\![\varphi]\!]_g$$
$$[\![\exists X : \sigma.\ \varphi]\!]_g = \exists X : \sigma.\ g\langle\sigma\rangle(X) \wedge [\![\varphi]\!]_g$$

The translation of a problem is given by $[\![\Phi]\!]_g = \mathrm{Ax} \cup \bigcup_{\varphi\in\Phi} [\![\varphi]\!]_g$, where Ax consists of the following *typing axioms*:

$$\forall\bar{\alpha}.\ \bar{X} : \bar{\sigma}.\ \left(\bigwedge_j\ g\langle\sigma_j\rangle(X_j)\right) \to g\langle\sigma\rangle(f\langle\bar{\alpha}\rangle(\bar{X})) \qquad \text{for } f : \forall\bar{\alpha}.\ \bar{\sigma} \to \sigma \in \mathcal{F}$$
$$\forall\alpha.\ \exists X : \alpha.\ g\langle\alpha\rangle(X)$$

The last axiom witnesses inhabitation of every type. It is necessary for completeness, in case some of the types do not appear in the result type of any function symbol.

Example 7. The g encoding translates the algebraic list problem of Example 2 into the following:

$$\forall A.\ g(\mathsf{list}(A), \mathsf{nil}(A))$$
$$\forall A, X, Xs.\ g(A, X) \wedge g(\mathsf{list}(A), Xs) \to g(\mathsf{list}(A), \mathsf{cons}(X, Xs))$$
$$\forall A, Xs.\ g(\mathsf{list}(A), Xs) \to g(A, \mathsf{hd}(Xs))$$
$$\forall A, Xs.\ g(\mathsf{list}(A), Xs) \to g(\mathsf{list}(A), \mathsf{tl}(Xs))$$
$$\forall A.\ \exists X.\ g(A, X)$$
$$\forall A, X, Xs.\ g(A, X) \wedge g(\mathsf{list}(A), Xs) \to \mathsf{nil}(A) \not\approx \mathsf{cons}(X, Xs)$$
$$\forall A, Xs.\ g(\mathsf{list}(A), Xs) \to$$
$$\quad Xs \approx \mathsf{nil}(A) \vee$$
$$\quad (\exists Y, Ys.\ g(A, Y) \wedge g(\mathsf{list}(A), Ys) \wedge Xs \approx \mathsf{cons}(Y, Ys))$$

$$\forall A, X, Xs.\ \mathsf{g}(A, X) \wedge \mathsf{g}(\mathsf{list}(A), Xs) \rightarrow \mathsf{hd}(\mathsf{cons}(X, Xs)) \approx X \wedge \mathsf{tl}(\mathsf{cons}(X, Xs)) \approx Xs$$
$$\exists X, Y, Xs, Ys.\ \mathsf{g}(\mathsf{b}, X) \wedge \mathsf{g}(\mathsf{b}, Y) \wedge \mathsf{g}(\mathsf{list}(\mathsf{b}), Xs) \wedge \mathsf{g}(\mathsf{list}(\mathsf{b}), Ys) \wedge \mathsf{cons}(X, Xs) \approx \mathsf{cons}(Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)$$

In this and later examples, we push guards inside past quantifiers and group them in a conjunction to enhance readability.

The typing axioms must in general be guarded. Without the guard, any cons, hd, or tl term could be typed with any type, defeating the purpose of the guard predicates.

4 Monotonicity-Based Type Encodings—The Monomorphic Case

Type tags and guards considerably increase the size of the problems passed to the automatic provers, with a dramatic impact on their performance. Most of the clutter can be removed by inferring monotonicity and soundly erasing type information based on the monotonicity analysis. Informally, a monotonic formula is one where, for any model of that formula, we can increase the size of the model while preserving satisfiability.

We focus on the monomorphic case, where the input problem contains no type variables or polymorphic symbols. Many of our definitions nonetheless handle the polymorphic case gracefully so that they can be reused in Section 5.

Before we start, let us define variants of the traditional t and g encodings that operate on monomorphic problems. The monomorphic encodings $\widetilde{\mathsf{t}}$ and $\widetilde{\mathsf{g}}$ coincide with t and g except that the polymorphic function $\mathsf{t}\langle\sigma\rangle(t)$ and predicate $\mathsf{g}\langle\sigma\rangle(t)$ are replaced by type-indexed families of unary functions $\mathsf{t}_\sigma(t)$ and predicates $\mathsf{g}_\sigma(t)$, as is customary in the literature [Wick and McCune, 1989, §4].

4.1 Monotonicity

We recap the definition of monotonicity from Paper 3.

Definition 14 (Finite Monotonicity)**.** Let σ be a ground type and Φ be a problem. The type σ is *finitely monotonic* in Φ if for all models $\mathcal{M}$ of Φ such that $[\![\sigma]\!]^{\mathcal{M}}$ is finite, there exists a model $\mathcal{M}'$ where $\left|[\![\sigma]\!]^{\mathcal{M}'}\right| =$

$\left|[\![\sigma]\!]^{\mathcal{M}}\right| + 1$ and $\left|[\![\tau]\!]^{\mathcal{M}'}\right| = \left|[\![\tau]\!]^{\mathcal{M}}\right|$ for all $\tau \neq \sigma$. The problem Φ is *finitely monotonic* if all its types are monotonic.

We propose a more permissive criterion, infinite monotonicity, that considers sets of types. Our definition assumes that all models are countable, but is easy to generalise to uncountable models.

Definition 15 (Infinite Monotonicity)**.** Let S be a set of ground types and Φ be a problem. The set S is (*infinitely*) *monotonic* in Φ if for all models $\mathcal{M}$ of Φ, there exists a model $\mathcal{M}'$ such that for all ground types σ, $[\![\sigma]\!]^{\mathcal{M}}$ is infinite if $\sigma \in S$ and $\left|[\![\sigma]\!]^{\mathcal{M}'}\right| = \left|[\![\sigma]\!]^{\mathcal{M}}\right|$ otherwise. A type σ is (*infinitely*) *monotonic* if $\{\sigma\}$ is monotonic. The problem Φ is (*infinitely*) *monotonic* if all its types, taken together, are monotonic.

Full type erasure is sound for monomorphic, monotonic problems. The intuition is that a model of such a problem can be extended into a model where all types are interpreted as sets of the same cardinality, which can be merged to yield an untyped model.

Example 8. The monkey village of Example 1 is infinitely monotonic because any model with finitely many bananas can be extended to a model with infinitely many, and any model with infinitely many bananas and finitely many monkeys can be extended to one where monkeys and bananas have the same infinite cardinality (cf. Example 4).

The next result allows us to focus on infinite monotonicity. Because of it, we will refer to infinite monotonicity as just *monotonicity* from now on.

Theorem 4 (Subsumption of Finite Monotonicity)**.** *Let σ be a ground type and Φ be a problem. If σ is finitely monotonic in Φ, then σ is infinitely monotonic in Φ.*

Proof. Assume that $\mathcal{M}$ is a model of Φ: we will find a model of Φ where $[\![\sigma]\!]$ is infinite and all other types have the same cardinality as in $\mathcal{M}$.

Let $\Phi\prime$ be Φ extended with axioms that state that for all types $\tau \neq \sigma$, the cardinality of τ is the same as in $\mathcal{M}$. Then $\Phi\prime$ remains finitely monotonic, and $\mathcal{M}$ is a model of it. It remains to show that there is a model of $\Phi\prime$ where $[\![\sigma]\!]$ is infinite.

Let K be the set of cardinality constraints $\left\{\left|[\![\sigma]\!]\right| \geqslant k \mid k \in \mathbb{N}\right\}$. Any finite subset of K asserts that $\left|[\![\sigma]\!]\right| \geqslant k$ for some $k \in \mathbb{N}$, while K as a whole asserts that $\left|[\![\sigma]\!]\right| \geqslant k$ for *all* k, i.e., $[\![\sigma]\!]$ is infinite.

Since σ is finitely monotonic in $\Phi\prime$, there are models of $\Phi\prime$ where $[\![\sigma]\!]$ is arbitrarily large (but finite). This means that $\Phi\prime$ taken together

with any finite subset of K is satisfiable. By compactness, $\Phi\prime \cup K$ as a whole must be satisfiable. But K is only satisfiable when $[\![t]\!]$ is infinite, so in this model $[\![t]\!]$ must be infinite. □

We need to introduce a few lemmas before we can reestablish the key result of Paper 4 for our notion of monotonicity.

Lemma 4 (Same-Cardinality Erasure). *Let Φ be a monomorphic problem. If Φ has a model where all domains have cardinality k, $[\![\Phi]\!]_e$ has a model where the unique domain has cardinality k.*

Proof. See either Theorem 1 in Bouillaguet et al. [2007, §4] or Lemma 1 in Paper 3. □

Lemma 5 (Downward Löwenheim–Skolem). *If a problem Φ has a model where all domains are infinite, it also has a model where all domains are countably infinite.*

Proof. Let $\Phi\prime$ be Φ extended with axioms that state that all domains are infinite. If Φ has a model where all domains are infinite, then $\Phi\prime$ is satisfiable. Since $\Phi\prime$ has a model, it has a Herbrand model. In a Herbrand model, all domains are finite or countable. But since a model of $\Phi\prime$ can have no finite domains, all domains of the Herbrand model of $\Phi\prime$ must be countably infinite. □

Theorem 5 (Monotonic Erasure). *Full type erasure is sound for monotonic monomorphic problems.*

Proof. Let Φ be such a problem. If Φ is satisfiable, Φ has a model where all domains are infinite by definition of monotonicity. By Lemma 5, it also has a model where all domains are countably infinite. Hence, $[\![\Phi]\!]_e$ is satisfiable by Lemma 4. □

The definition in terms of sets of types makes infinite monotonicity more powerful than finite monotonicity, but it is often more convenient to focus on single types and combine the results. The following lemma permits precisely such combinations. A consequence is that, if σ is monotonic in Φ for all types σ, then Φ is monotonic.

Lemma 6 (Monotonicity Preservation by Union). *Let $\mathcal{S}$ be a set of sets of ground types and Φ be a problem. If every $S \in \mathcal{S}$ is monotonic in Φ, then $\bigcup \mathcal{S}$ is monotonic in Φ.*

Proof. If $\mathcal{S}$ is empty or has 1 element, the lemma is trivial.

Suppose $|\mathcal{S}| = 2$, that is $\mathcal{S} = \{\sigma, \tau\}$. Then given a model of Φ, we first use σ's monotonicity to get a model where $[\![\sigma]\!]$ is infinite, and all other domains are unchanged. Then we use τ's monotonicity to get a model where $[\![\sigma]\!]$ and $[\![\tau]\!]$ are infinite, and all other domains are unchanged. This is the required model. By induction, this establishes the lemma for finite $\mathcal{S}$.

If $\mathcal{S}$ is infinite, we use the same compactness argument as in Theorem 4. Suppose $\mathcal{M}$ is a model of Φ. Let the set of formulas K consist of, for all types $\sigma \in \mathcal{S}$, the statement that $[\![\sigma]\!]$ is infinite, and for all types $\sigma \notin \mathcal{S}$, the statement that the cardinality of $[\![\sigma]\!]$ is the same as in $\mathcal{M}$.

To show that S is monotonic in Φ, we need to show that $\Phi \cup \mathsf{K}$ is satisfiable. Since any finite subset of $\mathcal{S}$ is monotonic in Φ, the union of Φ and any finite subset of K is satisfiable. Hence, by compactness, $\Phi \cup \mathsf{K}$ is satisfiable. □

4.2 Monotonicity Inference

Paper 3 gave a simple calculus to infer finite monotonicity for monomorphic first-order logic. The definition below generalises it from clause normal form to negation normal form. The generalisation is straightforward; we present it because we later adapt it to polymorphism. The calculus is based on the observation that a type σ must be monotonic if the problem expressed in NNF contains no positive literal of the form $X^\sigma \approx t$ or $t \approx X^\sigma$, where X is universal. We call such an occurrence of X a naked occurrence. Naked variables are the only way to express upper bounds on the cardinality of types in first-order logic.

Definition 16 (Naked Variable)**.** The set of *naked variables* $\mathrm{NV}(\varphi)$ of a formula φ is defined as follows:

$$\begin{aligned}
\mathrm{NV}(\mathsf{p}\langle\bar{\sigma}\rangle(\bar{t})) &= \emptyset \\
\mathrm{NV}(\neg\, \mathsf{p}\langle\bar{\sigma}\rangle(\bar{t})) &= \emptyset \\
\mathrm{NV}(t_1 \approx t_2) &= \{t_1, t_2\} \cap \mathcal{V} \\
\mathrm{NV}(t_1 \not\approx t_2) &= \emptyset \\
\mathrm{NV}(\varphi_1 \wedge \varphi_2) &= \mathrm{NV}(\varphi_1) \cup \mathrm{NV}(\varphi_2) \\
\mathrm{NV}(\varphi_1 \vee \varphi_2) &= \mathrm{NV}(\varphi_1) \cup \mathrm{NV}(\varphi_2) \\
\mathrm{NV}(\forall X : \sigma.\ \varphi) &= \mathrm{NV}(\varphi) \\
\mathrm{NV}(\exists X : \sigma.\ \varphi) &= \mathrm{NV}(\varphi) - \{X\} \\
\mathrm{NV}(\forall \alpha.\ \varphi) &= \mathrm{NV}(\varphi)
\end{aligned}$$

Notice the equation for the $\exists$ case: existential variables are never naked.

Paper 3 gave a second, more powerful calculus to detect predicates that act as guards for naked variables. Whilst the calculus proved successful on a subset of the TPTP benchmarks [Sutcliffe, 2009], we assessed its suitability on about 1000 problems generated by Sledgehammer and found no improvement on the simple calculus. We restrict our attention to the first calculus.

Variables of types other than σ are irrelevant when inferring whether σ is monotonic; a variable is problematic only if it occurs naked and has type σ. Annoyingly, a single naked variable of type σ, such as X on the right-hand side of the equation $\mathsf{hd_b}(\mathsf{cons_b}(X, Xs)) \approx X$ from Example 3, will cause us to classify σ as possibly non-monotonic.

We regain some precision by extending the calculus with an infinity analysis: trivially, all types with no finite models are monotonic. Abstracting over the specific analysis used to detect infinite types (e.g. Infinox [Claessen and Lillieström, 2011]), we fix a set $\mathrm{Inf}(\Phi)$ of types whose interpretations are guaranteed to be infinite in all models of Φ. The monotonicity calculus takes $\mathrm{Inf}(\Phi)$ into account.

Definition 17 (Monotonicity Calculus $\rhd$). Let Φ be a monomorphic problem over $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$. A judgement $\sigma \rhd \varphi$ indicates that the ground type σ is inferred monotonic in $\varphi \in \Phi$. The *monotonicity calculus* consists of the following rules:

$$\frac{\sigma \in \mathrm{Inf}(\Phi)}{\sigma \rhd \varphi} \qquad \frac{\mathrm{NV}(\varphi) \cap \{X \mid X \text{ has type } \sigma\} = \emptyset}{\sigma \rhd \varphi}$$

$\mathrm{Inf}(\Phi)$ is a set of ground types over $\mathcal{K}$ (the *infinite types*) which all models of Φ interpret as infinite sets; We write $\sigma \rhd \varphi$ to indicate that the judgement is derivable and $\sigma \not\rhd \varphi$ if it is not derivable. These notations are lifted to problems Φ, with $\sigma \rhd \Phi$ if and only if $\sigma \rhd \varphi$ for all $\varphi \in \Phi$.

Theorem 6 (Soundness of $\rhd$). *Let Φ be a monomorphic problem. If $\sigma \rhd \Phi$, then σ is monotonic in Φ.*

Proof. Given a model $\mathcal{M}$ where σ is interpreted as D, we construct a model $\mathcal{M}'$ where σ is interpreted as the infinite set $D \uplus D'$. If $D' \neq \emptyset$, we choose an arbitrary representative element $a \in D$ and extend the interpretations of all functions and predicates so that they coincide on a and each $a' \in D'$. For problems with no equality over σ, $\mathcal{M}'$ is obviously a model. Equality atoms between non-variables $f(\ldots) \approx g(\ldots)$ do not compromise the construction, because the new element a' cannot

be in the functions' ranges. Nor do negative equality literals pose any difficulties, because although the cases $a \not\approx a'$, $a' \not\approx a$, and $a' \not\approx a'$ are possible by instantiating naked variables with a', they are weaker than the cases where the same variables are instantiated with a, which themselves must be true for $\mathcal{M}$ to be a model. □

Even though monotonicity is defined on sets of types, we allow ourselves to write that σ is monotonic if $\sigma \rhd \Phi$ and possibly non-monotonic if $\sigma \not\rhd \Phi$.

4.3 Encoding Non-monotonic Types

Monotonic types can be soundly erased when translating to untyped first-order logic, by Theorem 5. Non-monotonic types in general cannot. The idea of Paper 3 is that adding sufficiently many protectors to a non-monotonic problem will make it monotonic, after which its types can be erased. Thus the following general two-stage procedure translates monomorphic problems to untyped first-order logic:

1. Introduce protectors (tags or guards) without erasing any types:
   a) Infer monotonicity to identify the possibly non-monotonic types in the problem.
   b) Introduce protectors for universal variables of possibly non-monotonic types.
   c) If necessary, generate *typing axioms* for any function symbol whose result type is possibly non-monotonic, to make it possible to remove protectors.
2. Erase all the types.

The purpose of stage 1 is to make the problem monotonic while preserving satisfiability. This paves the way for the sound type erasure of stage 2. The following lemmas will help us prove such two-stage encodings correct.

Lemma 7 (Correctness Conditions). *Let Φ be a monomorphic problem, and let x be a monomorphic encoding. The problems Φ and $[\![\Phi]\!]_{x;e}$ are equisatisfiable provided the following conditions hold:*

MONO: *$[\![\Phi]\!]_x$ is monotonic.*

SOUND: *If Φ is satisfiable, so is $[\![\Phi]\!]_x$.*

COMPLETE: *If* $[\![\Phi]\!]_x$ *is satisfiable, so is* Φ.

Proof. Immediate from Theorems 2 and 5. □

Lemma 8 (Submodel). *Let* Φ *be a problem, let* $\mathcal{M}$ *be a model of* Φ*, and let* $\mathcal{M}'$ *be a substructure of* $\mathcal{M}$ *(i.e. a structure constructed from* $\mathcal{M}$ *by removing some domain elements while leaving the interpretations of functions and predicates intact for the remaining elements). This* $\mathcal{M}'$ *is a model of* Φ *provided that it does not remove any witness for an existential variable.*

Proof. Suppose Φ is skolemised. If it is *false* in $\mathcal{M}'$ then there must be a valuation—an assignment of domain elements to the variables of Φ—which makes it false. This same valuation makes Φ false in $\mathcal{M}$.

If Φ is not skolemised, let Ψ be its skolemisation. From $\mathcal{M}$ we can construct a model $\mathcal{M}_\Psi$ of Ψ by giving interpretations to all the Skolem functions in Ψ. We remove domain elements from $\mathcal{M}$ to get $\mathcal{M}'$; we can remove the same domain elements from $\mathcal{M}_\Psi$ to get $\mathcal{M}'_\Psi$. This gives us a substructure of $\mathcal{M}_\Psi$ as long as we do not remove any domain elements that are in the image of a Skolem function in $\mathcal{M}_\Psi$, which is to say, we do not remove any witness for an existential variable in $\mathcal{M}$. By the result above, $\mathcal{M}'_\Psi$ is a model of Ψ, hence $\mathcal{M}'$ is a model of Φ. □

4.4 Monotonicity-Based Type Tags

The monotonicity-based encoding $\widetilde{t?}$ specialises this procedure for tags. It is similar to the traditional encoding $\widetilde{t}$ (the monomorphic version of t), except that it omits the tags for types that are inferred monotonic. By wrapping all naked variables (in fact, all terms) of possibly non-monotonic types in a function term, stage 1 yields a monotonic problem.

Definition 18 (Lightweight Tags $\widetilde{t?}$). The *monomorphic lightweight type tags* encoding $\widetilde{t?}$ translates a monomorphic problem Φ over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \{t^1_\sigma \mid \sigma \text{ is ground over } \mathcal{K}\}, \mathcal{P}')$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for e. It is defined as $[\![\ ]\!]_{\widetilde{t?};e}$, where

$$[\![f(\bar{t})]\!]_{\widetilde{t?}} = \lfloor f([\![\bar{t}]\!]_{\widetilde{t?}}) \rfloor \quad [\![X]\!]_{\widetilde{t?}} = \lfloor X \rfloor \quad \text{with} \quad \lfloor t^\sigma \rfloor = \begin{cases} t & \text{if } \sigma \rhd \Phi \\ t_\sigma(t) & \text{otherwise} \end{cases}$$

Example 9. For the algebraic list problem of Example 3, the type list_b is monotonic by virtue of being infinite, whereas b cannot be inferred

monotonic. The $\widetilde{\text{t}}$? encoding of the problem follows:

$$\begin{aligned}
&\forall X, Xs.\ \mathsf{nil}_b \not\approx \mathsf{cons}_b(t_b(X), Xs)\\
&\forall Xs.\ Xs \approx \mathsf{nil}_b \vee (\exists Y, Ys.\ Xs \approx \mathsf{cons}_b(t_b(Y), Ys))\\
&\forall X, Xs.\\
&\quad t_b(\mathsf{hd}_b(\mathsf{cons}_b(t_b(X), Xs))) \approx t_b(X) \wedge\\
&\quad \mathsf{tl}_b(\mathsf{cons}_b(t_b(X), Xs)) \approx Xs\\
&\exists X, Y, Xs, Ys.\\
&\quad \mathsf{cons}_b(t_b(X), Xs) \approx \mathsf{cons}_b(t_b(Y), Ys) \wedge\\
&\quad (t_b(X) \not\approx t_b(Y) \vee Xs \not\approx Ys)
\end{aligned}$$

The $\widetilde{\text{t}}$? encoding treats all variables of the same type uniformly. Hundreds of axioms can suffer because of one unhappy formula that uses a type non-monotonically (or in a way that cannot be inferred monotonic). To address this, we introduce a lighter encoding: if a universal variable does not occur naked in a formula, its tag can safely be omitted.[1]

Our novel encoding $\widetilde{\text{t}}$?? protects only naked variables and introduces equations $t_\sigma(f(\overline{X})^\sigma) \approx f(\overline{X})$ to add or remove tags around each function symbol f whose result type σ is possibly non-monotonic, and similarly for existential variables.

Definition 19 (Featherweight Tags $\widetilde{\text{t}}$??)**.** The *monomorphic featherweight type tags* encoding $\widetilde{\text{t}}$?? translates a monomorphic problem Φ over Σ into an untyped problem over Σ', where Σ' is as for $\widetilde{\text{t}}$?. It is defined as $[\![\ ]\!]_{\widetilde{\text{t}}??;e}$, where

$$\begin{aligned}
[\![t_1 \approx t_2]\!]_{\widetilde{\text{t}}??} &= \lfloor [\![t_1]\!]_{\widetilde{\text{t}}??} \rfloor \approx \lfloor [\![t_2]\!]_{\widetilde{\text{t}}??} \rfloor\\
[\![\exists X : \sigma.\ \varphi]\!]_{\widetilde{\text{t}}??} &= \exists X : \sigma. \begin{cases} [\![\varphi]\!]_{\widetilde{\text{t}}??} & \text{if } \sigma \rhd \Phi\\ t_\sigma(X) \approx X \wedge [\![\varphi]\!]_{\widetilde{\text{t}}??} & \text{otherwise} \end{cases}
\end{aligned}$$

with

$$\lfloor t^\sigma \rfloor = \begin{cases} t & \text{if } \sigma \rhd \Phi \text{ or } t \text{ is not a universal variable}\\ t_\sigma(t) & \text{otherwise} \end{cases}$$

The encoding is complemented by typing axioms:

$\forall \overline{X} : \bar{\sigma}.\ t_\sigma(f(\overline{X})) \approx f(\overline{X})$	for $f : \bar{\sigma} \to \sigma \in \mathcal{F}$ such that $\sigma \not\rhd \Phi$
$\exists X : \sigma.\ t_\sigma(X) \approx X$	for $\sigma \not\rhd \Phi$ that is not the result type of a symbol in $\mathcal{F}$

[1] This is related to the observation that only paramodulation from or into a variable can cause ill-typed instantiations in a resolution prover [Wick and McCune, 1989, §4].

The side condition for the last axiom is a minor optimisation: it avoids asserting that σ is inhabited if the symbols in $\mathcal{F}$ already witness σ's inhabitation.

Example 10. The $\widetilde{\mathsf{t}}??$ encoding of Example 3 requires fewer tags than $\widetilde{\mathsf{t}}?$, at the cost of more type information (for hd and the existential variables of type b):

$$\forall \mathrm{Xs}.\ \mathsf{t}_\mathsf{b}(\mathsf{hd}_\mathsf{b}(\mathrm{Xs})) \approx \mathsf{hd}_\mathsf{b}(\mathrm{Xs})$$

$$\begin{array}{l}
\forall \mathrm{X}, \mathrm{Xs}.\ \mathsf{nil}_\mathsf{b} \not\approx \mathsf{cons}_\mathsf{b}(\mathrm{X}, \mathrm{Xs}) \\
\forall \mathrm{Xs}.\ \mathrm{Xs} \approx \mathsf{nil}_\mathsf{b} \vee (\exists \mathrm{Y}, \mathrm{Ys}.\ \mathsf{t}_\mathsf{b}(\mathrm{Y}) \approx \mathrm{Y} \wedge \mathrm{Xs} \approx \mathsf{cons}_\mathsf{b}(\mathrm{Y}, \mathrm{Ys})) \\
\forall \mathrm{X}, \mathrm{Xs}.\ \mathsf{hd}_\mathsf{b}(\mathsf{cons}_\mathsf{b}(\mathrm{X}, \mathrm{Xs})) \approx \mathsf{t}_\mathsf{b}(\mathrm{X}) \wedge \mathsf{tl}_\mathsf{b}(\mathsf{cons}_\mathsf{b}(\mathrm{X}, \mathrm{Xs})) \approx \mathrm{Xs} \\
\exists \mathrm{X}, \mathrm{Y}, \mathrm{Xs}, \mathrm{Ys}. \\
\quad \mathsf{t}_\mathsf{b}(\mathrm{X}) \approx \mathrm{X} \wedge \mathsf{t}_\mathsf{b}(\mathrm{Y}) \approx \mathrm{Y} \wedge \\
\quad \mathsf{cons}_\mathsf{b}(\mathrm{X}, \mathrm{Xs}) \approx \mathsf{cons}_\mathsf{b}(\mathrm{Y}, \mathrm{Ys}) \wedge (\mathrm{X} \not\approx \mathrm{Y} \vee \mathrm{Xs} \not\approx \mathrm{Ys})
\end{array}$$

Theorem 7 (Correctness of $\widetilde{\mathsf{t}}?$, $\widetilde{\mathsf{t}}??$). *The monomorphic type tags encodings* $\widetilde{\mathsf{t}}?$ *and* $\widetilde{\mathsf{t}}??$ *are correct.*

Proof. It suffices to show the three conditions of Lemma 7.

Mono: By Theorem 6, types are monotonic unless they are possibly finite and variables of their types occur naked in the original problem. Both encodings protect all such variables—$\widetilde{\mathsf{t}}??$ tags exactly these variables, while $\widetilde{\mathsf{t}}?$ tags even more terms. The typing axioms contain no naked variables. Since all types are monotonic, the encoded problem is monotonic by Lemma 6.

Sound: Given a model of Φ, we extend it to a model of $[\![\Phi]\!]_x$ by interpreting all type tags as the identity.

Complete: For $\widetilde{\mathsf{t}}?$, we prove the contrapositive: if $[\![\Phi]\!]_x$ is unsatisfiable then so is Φ. The problem $[\![\Phi]\!]_x$ is Φ transformed so that a type tag t_σ surrounds every term and subterm of non-monomorphic type σ. Take a contradictory set of ground instances of Φ; applying the same transformation to the terms in that set gives a contradictory set of ground instances of $[\![\Phi]\!]_x$.

For $\widetilde{\mathsf{t}}??$, we take a different approach. A model of $[\![\Phi]\!]_x$ is *canonical* if all tag functions t_σ are interpreted as the identity. From a canonical model, we obtain a model of Φ by the converse construction to Sound. It then suffices to prove that whenever there exists a model $\mathcal{M}$ of $[\![\Phi]\!]_x$, there exists a canonical model $\mathcal{M}'$.

To construct a canonical model, we take $\mathcal{M}$, and for each $\sigma \not\rhd \Phi$, we simply remove the model elements for which t_σ is not the identity. The typing axioms ensure that the substructure is well-defined: each tag function is the identity for at least one element and also for each element within the range of a non-tag function. The equations $\mathsf{t}_\sigma(X) \approx X$ generated for existential variables ensure that witnesses are preserved; by Lemma 8, $\mathcal{M}'$ is a model of Φ. □

The above proof goes through even if the generated problems contain more tags than are necessary to ensure monotonicity. Moreover, we may add further typing axioms to $[\![\Phi]\!]_x$—for example, equations $\mathsf{f}(\overline{U}, \mathsf{t}_\sigma(X), \overline{V}) \approx \mathsf{f}(\overline{U}, X, \overline{V})$ to add or remove tags around well-typed arguments of a symbol f, or the idempotence law $\mathsf{t}_\sigma(\mathsf{t}_\sigma(X)) \approx \mathsf{t}_\sigma(X)$—provided that they hold for canonical models and preserve monotonicity.

4.5 Monotonicity-Based Type Guards

The $\widetilde{\mathsf{g}}?$ and $\widetilde{\mathsf{g}}??$ encodings are defined analogously to $\widetilde{\mathsf{t}}?$ and $\widetilde{\mathsf{t}}??$ but using type guards. The $\widetilde{\mathsf{g}}?$ encoding omits the guards for types that are inferred monotonic, whereas $\widetilde{\mathsf{g}}??$ omits more guards that are not needed to make the intermediate problem monotonic.

Definition 20 (Lightweight Guards $\widetilde{\mathsf{g}}?$)**.** The *monomorphic lightweight type guards* encoding $\widetilde{\mathsf{g}}?$ translates a monomorphic problem Φ over Σ into an untyped problem over $\Sigma' = (\mathcal{F}', \mathcal{P}' \uplus \{\mathsf{g}^1_\sigma \mid \sigma \text{ is ground over } \mathcal{K}\})$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for $\mathtt{e}$. It is defined as $[\![\ ]\!]_{\widetilde{\mathsf{g}}?;\mathtt{e}}$, where

$$[\![\forall X : \sigma.\ \varphi]\!]_{\widetilde{\mathsf{g}}?} = \forall X : \sigma.\ \begin{cases} [\![\varphi]\!]_{\widetilde{\mathsf{g}}?} & \text{if } \sigma \rhd \Phi \\ \mathsf{g}_\sigma(X) \to [\![\varphi]\!]_{\widetilde{\mathsf{g}}?} & \text{otherwise} \end{cases}$$

$$[\![\exists X : \sigma.\ \varphi]\!]_{\widetilde{\mathsf{g}}?} = \exists X : \sigma.\ \begin{cases} [\![\varphi]\!]_{\widetilde{\mathsf{g}}?} & \text{if } \sigma \rhd \Phi \\ \mathsf{g}_\sigma(X) \wedge [\![\varphi]\!]_{\widetilde{\mathsf{g}}?} & \text{otherwise} \end{cases}$$

The encoding is complemented by typing axioms:

$$\begin{array}{ll} \forall \bar{X} : \bar{\sigma}.\ \mathsf{g}_\sigma(\mathsf{f}(\bar{X})) & \text{for } \mathsf{f} : \bar{\sigma} \to \sigma \in \mathcal{F} \text{ such that } \sigma \not\rhd \Phi \\ \exists X : \sigma.\ \mathsf{g}_\sigma(X) & \text{for } \sigma \not\rhd \Phi \text{ that is not the result type of a symbol in } \mathcal{F} \end{array}$$

Example 11. The $\widetilde{\mathsf{g}}?$ encoding of Example 3 is as follows:

$$\forall Xs.\ \mathsf{g}_\mathsf{b}(\mathsf{hd}_\mathsf{b}(Xs))$$

$$\forall X,\ Xs.\ \mathsf{g}_\mathsf{b}(X) \to \mathsf{nil}_\mathsf{b} \not\approx \mathsf{cons}_\mathsf{b}(X, Xs)$$

$$\forall Xs.\ Xs \approx \mathsf{nil}_b \vee (\exists Y,\ Ys.\ \mathsf{g}_b(Y) \wedge Xs \approx \mathsf{cons}_b(Y,\ Ys))$$
$$\forall X : b,\ Xs.\ \mathsf{g}_b(X) \rightarrow$$
$$\mathsf{hd}_b(\mathsf{cons}_b(X,\ Xs)) \approx X \wedge \mathsf{tl}_b(\mathsf{cons}_b(X,\ Xs)) \approx Xs$$
$$\exists X,\ Y,\ Xs,\ Ys.\ \mathsf{g}_b(X) \wedge \mathsf{g}_b(Y) \wedge$$
$$\mathsf{cons}_b(X,\ Xs) \approx \mathsf{cons}_b(Y,\ Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)$$

Notice that the tl_b equation is needlessly in the scope of the guard. The encoding is more precise if the problem is clausified.

By leaving Xs unconstrained, the typing axiom gives a type to some ill-typed terms, such as $\mathsf{hd}_b(0^{nat})$. Intuitively, this is safe because such terms cannot be used to prove anything useful that could not be proved by a well-typed typed. What matters is that well-typed terms are associated with their correct type.

Our novel encoding $\widetilde{\mathsf{g}}??$ omits the guards for variables that do no occur naked, regardless of whether they are of a monotonic type.

Definition 21 (Featherweight Guards $\widetilde{\mathsf{g}}??$)**.** The *monomorphic featherweight type guards* encoding $\widetilde{\mathsf{g}}??$ is identical to the lightweight encoding $\widetilde{\mathsf{g}}?$ except that the condition "if $\sigma \rhd \Phi$" in the $\forall$ case is weakened to "if $\sigma \rhd \Phi$ or $X \notin \mathrm{NV}(\varphi)$".

Example 12. The $\widetilde{\mathsf{g}}??$ encoding of the algebraic list problem is identical to $\widetilde{\mathsf{g}}?$ except that the $\mathsf{nil}_b \not\approx \mathsf{cons}_b$ axiom does not have any guard.

Theorem 8 (Correctness of $\widetilde{\mathsf{g}}?$, $\widetilde{\mathsf{g}}??$)**.** *The monomorphic type guards encodings* $\widetilde{\mathsf{g}}?$ *and* $\widetilde{\mathsf{g}}??$ *are correct.*

Proof. It suffices to show the three conditions of Lemma 7.

Mono: By Theorem 6, types are monotonic unless they are possibly finite and variables of their types occur naked in the original problem. Both encodings guard all such variables—$\widetilde{\mathsf{g}}??$ guards exactly those variables, while $\widetilde{\mathsf{g}}?$ guards more. The typing axioms contain no naked variables. We cannot use Theorem 6 directly, because guarding a naked variable does not make it less naked, but we can generalise the proof slightly to exploit the guards. Given a model $\mathcal{M}$ of $[\![\Phi]\!]_x$ where each σ is interpreted as D_σ, we construct a model $\mathcal{M}'$ of $[\![\Phi]\!]_x$ where σ is interpreted as $D_\sigma \uplus D'_\sigma$. We choose a representative $a \in D_\sigma$ and extend the interpretations of all symbols so that they coincide on a and each $a' \in D'_\sigma$ except that $\mathsf{g}_\sigma(a')$ is interpreted as false. This is compatible with the typing axioms (since a' never occurs in the range of a function) and effectively prevents a' from arising naked.

SOUND: Given a model of Φ, we extend it to a model of $[\![\Phi]\!]_x$ by interpreting all type guards as the true predicate (the predicate that is true everywhere).

COMPLETE: A model of $[\![\Phi]\!]_x$ is *canonical* if all guards are interpreted as the true predicate. From a canonical model, we obtain a model of Φ by the converse construction to SOUND. It then suffices to prove that whenever there exists a model $\mathcal{M}$ of $[\![\Phi]\!]_x$, there exists a canonical model $\mathcal{M}'$. We construct $\mathcal{M}'$ by removing the domain elements from $\mathcal{M}$ that do not satisfy their guard predicate. The typing axioms ensure that the substructure is well-defined: each guard predicate is satisfied by at least one element and each function is satisfy its predicate. The guards $g_\sigma(X)$ generated for existential variables ensure that witnesses are preserved, as required by Lemma 8. □

A simpler but less instructive way to prove MONO is to observe that the second monotonicity calculus of Paper 3 can infer monotonicity of all problems generated by $\widetilde{g}$, $\widetilde{g}?$, and $\widetilde{g}??$.

4.6 Finite Monomorphisation

Section 5 will show how to translate polymorphic types soundly and completely. If we are willing to sacrifice completeness, an easy way to extend $\widetilde{t}?$, $\widetilde{t}??$, $\widetilde{g}?$, and $\widetilde{g}??$ to polymorphism is to perform *finite monomorphisation* on the polymorphic problem:

1. Heuristically instantiate all type variables with suitable ground types, taking as many copies of the formulas as desired.

2. Map each ground occurrence $s\langle\bar{\alpha}\rho\rangle$ of a polymorphic symbol $s : \forall\bar{\alpha}.\ \bar{\sigma} \to \sigma$ to a fresh monomorphic symbol $s_{\bar{\alpha}\rho} : \lfloor\bar{\sigma}\rho\rfloor \to \lfloor\sigma\rho\rfloor$, where ρ is a ground type substitution (a function from type variables to ground types) and $\lfloor\ \rfloor$ is an injection from ground types to nullary type constructors (e.g. $\{b \mapsto b, \mathtt{list}(b) \mapsto \mathtt{list_b}\}$).

Finite monomorphisation is generally incomplete [Bobot and Paskevich, 2011, §2] and often overlooked in the literature, but by eliminating type variables it considerably simplifies the generated formulas, leading to very efficient encodings. It also provides a simple and effective way to exploit the native support for monomorphic types in some automatic provers.

5 Complete Monotonicity-Based Encoding of Polymorphism

Finite monomorphisation is simple and effective, but its incompleteness can be a cause for worry, and its non-modular nature makes it unsuitable for some applications that need to export an entire polymorphic theory independently of any conjecture. Here we adapt the monotonicity calculus, the type encodings, and the proofs from the previous section to a polymorphic setting.

5.1 Type Arguments to Constructors

We start with a brief digression. With monotonicity-based encoding schemes, type arguments are needed to distinguish instances of polymorphic symbols. These additional arguments introduce clutter, which we can eliminate in some cases. The result is an optimised variant $\mathsf{a}^{\mathsf{ctor}}$ of the type arguments encoding a, which will serve as the foundation for $\mathsf{t?}$, $\mathsf{t??}$, $\mathsf{g?}$, and $\mathsf{g??}$.

Consider a type $\mathsf{sum}(\alpha, \beta)$ that is axiomatised to be freely constructed by $\mathsf{inl} : \alpha \to \mathsf{sum}(\alpha, \beta)$ and $\mathsf{inr} : \beta \to \mathsf{sum}(\alpha, \beta)$. Regardless of β, inl must be interpreted as an injection from α to $\mathsf{sum}(\alpha, \beta)$. For a fixed α, its interpretations for different β instances are isomorphic. As a result, it is safe to omit the type argument for β when encoding $\mathsf{inl}\langle\alpha, \beta\rangle$ and that for α in $\mathsf{inr}\langle\alpha, \beta\rangle$ and $\mathsf{nil}\langle\alpha\rangle : \mathsf{list}(\alpha)$. In general, the type arguments that can be omitted for constructors are precisely those that are non-inferrable in the sense of Definition 12. We call this encoding $\mathsf{a}^{\mathsf{ctor}}$. The encodings presented below exploit the fact that $[\![\Phi]\!]_{\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$ is equisatisfiable to Φ if Φ is monotonic.

Definition 22 (Constructor)**.** Given a polymorphic problem Φ over $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$, a set of *constructors* for Φ consists of symbols $\mathsf{c} : \forall\bar{\alpha}.\ \bar{\sigma} \to \mathsf{k}(\bar{\alpha}) \in \mathcal{F}$ such that the injectivity law

$$\forall\bar{X}, \bar{Y}.\ \mathsf{c}(\bar{X}) \approx \mathsf{c}(\bar{Y}) \to \bar{X} \approx \bar{Y}$$

holds in all models of Φ, and for each pair of distinct constructors c, d whose result type involves the same type constructor, distinctness also hold:

$$\forall\bar{X}, \bar{Y}.\ \mathsf{c}(\bar{X}) \not\approx \mathsf{d}(\bar{Y})$$

Abstracting over the specific analysis used to detect constructors, we fix a set of constructors $\mathrm{Ctor}(\Phi) \subseteq \mathcal{F}$.

Definition 23 (Constructor-Needed Type Argument)**.** Let Φ be a polymorphic problem over $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$, and let $s : \forall \alpha_1, \ldots, \alpha_m.\ \bar{\sigma} \to \sigma \in \mathcal{F} \uplus \mathcal{P}$. The ith type argument is *constructor-needed* if either $s \notin \mathrm{Ctor}(\Phi)$ or α_i is inferrable. Given a list $\bar{z}$, $\mathrm{ctor}_s(\bar{z})$ denotes the sublist corresponding to the constructor-needed type arguments.

The notion of constructor-needed type arguments induces, via Definition 9, a type encoding $\mathsf{a}^{\mathsf{ctor}}$ that is lighter than a and that enjoys interesting properties.

Example 13. The $\mathsf{a}^{\mathsf{ctor}}$ encoding of the polymorphic algebraic list problem of Example 2 is identical to the a encoding (Example 5) except that nil replaces $\mathsf{nil}(A)$.

5.2 Monotonicity

Definition 15 and Lemmas 5, 8, and 6 cater for polymorphic problems. Other results from Section 4 must be adapted to the polymorphic case.

Definition 24 (Monotonicity of Polymorphic Type)**.** A polymorphic type is (*infinitely*) *monotonic* if all its ground instances are monotonic.

Theorem 9 (Monotonic Erasure)**.** *The constructor-needed type arguments encoding* $\mathsf{a}^{\mathsf{ctor}}$ *is sound for monotonic polymorphic problems.*

Proof. Let Φ be such a problem, and let $\mathcal{M}$ be a model of Φ. By Lemma 5, we may assume that all types in $\mathcal{M}$ are interpreted by the same infinite domain D. We first adjust the model $\mathcal{M}$ to synchronise the interpretation of the constructors, so that constructor instances whose inferrable type arguments coincide have semantics that coincide. The definition of inferrable type arguments, together with injectivity, ensures that the considered interpretations are isomorphic. From there, we can permute each type's domain elements to obtain true coincidence, proceeding one constructor at a time and relying on the distinctness property to ensure that $\mathcal{M}$ remains a model.

From this adjusted model $\mathcal{M}$, we construct a model $\mathcal{M}'$ of $[\![\Phi]\!]_{\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$ that interprets the encoded types as distinct elements of D. The function and predicate tables for $\mathcal{M}'$ are based on those from $\mathcal{M}$, with encoded type arguments corresponding to actual type arguments. For constructors, some type arguments may be missing in $\mathcal{M}'$, but they are irrelevant since the interpretations in $\mathcal{M}$ coincide. □

Lemma 9 (Correctness Conditions). *Let Φ be a polymorphic problem, and let x be a polymorphic encoding. The problems Φ and $[\![\Phi]\!]_{x;a^{ctor};e}$ are equisatisfiable provided the following conditions hold:*

MONO: *$[\![\Phi]\!]_x$ is monotonic.*

SOUND: *If Φ is satisfiable, so is $[\![\Phi]\!]_x$.*

COMPLETE: *If $[\![\Phi]\!]_x$ is satisfiable, so is Φ.*

Proof. Immediate from Theorems 3 and 9. □

5.3 Monotonicity Inference

The monotonicity inference of Section 4.2 must be adapted to the polymorphic setting. The calculus presented below captures the insight that a polymorphic type is monotonic if each of its common instances with the type of any naked variable is an instance of an infinite type.

Definition 25 (Monotonicity Calculus $\rhd$). Let Φ be a polymorphic problem. A judgement $\sigma \rhd \varphi$ indicates that the type σ is inferred monotonic in $\varphi \in \Phi$. The *monotonicity calculus* consists of the single rule

$$\frac{\forall X^{\tau} \in \mathsf{NV}(\varphi).\ \mathrm{mgu}(\sigma,\tau) \in \mathrm{Inf}^{*}(\varphi)}{\sigma \rhd \varphi}$$

where $\mathrm{mgu}(\sigma,\tau)$ is the most general unifier of σ and τ, and $\mathrm{Inf}^{*}(\varphi)$ consists of all instances of all types in $\mathrm{Inf}(\varphi)$.

Remark 1. Although type variables occurring in declarations and formulas are bound by a $\forall$ quantifier, for monotonicity inference polymorphic types $\sigma[\bar{\alpha}]$, where $\bar{\alpha}$ are the type variables occurring in σ, are viewed as being implicitly bound (i.e. "$\forall\bar{\alpha}.\ \sigma[\bar{\alpha}]$" to abuse notation). Type variables are assumed to be fresh in distinct types, comparison between types is modulo α-renaming, and substitution is capture-avoiding. Thus, we have $\mathtt{list}(\alpha) = \mathtt{list}(\beta)$ (since "$\forall\alpha.\ \mathtt{list}(\alpha) = \forall\beta.\ \mathtt{list}(\beta)$"), and $\mathtt{map}(\alpha, d)$ can be unified with $\mathtt{map}(c, \alpha)$ by renaming the second α to β and applying $\rho = \{\alpha \mapsto c,\ \beta \mapsto d\}$.

Our proof strategy is to reduce the polymorphic case to the already proved monomorphic case. Our Herbrandian motto is,

> A polymorphic formula is equisatisfiable to the (generally infinite) set of its monomorphic instances.

This complete form of monomorphisation is not to be confused with the finitary, heuristic monomorphisation algorithm presented in Section 4.6.

Theorem 10 (Soundness of $\rhd$). *Let Φ be a polymorphic problem. If $\sigma \rhd \Phi$, then σ is monotonic in Φ.*

Proof. We consider an unknown but fixed monomorphic instance of the rule and justify it by one of the two rules of the monomorphic version of the calculus, proved sound in Theorem 6. This is sufficient because the rule honours the abstract view of polymorphic types as sets of ground types, without inspecting their concrete structure. For monomorphic σ and φ, suppose that the polymorphic monotonicity calculus judges that $\sigma \rhd \varphi$. By Definition 25, there is no $X^\tau \in \mathrm{NV}(\varphi)$ such that σ and τ are unifiable to $\beta \notin \mathrm{Inf}^*(\varphi)$. Since both σ and φ are monomorphic, this states that for all $X^\sigma \in \mathrm{NV}(\varphi)$, $\sigma \in \mathrm{Inf}(\varphi)$, i.e. either $\mathrm{NV}(\varphi)$ contains no variables of type σ, or $\sigma \in \mathrm{Inf}(\varphi)$. In both cases, a rule of the monomorphic monotonicity calculus applies. □

5.4 Monotonicity-Based Type Tags

The polymorphic t? encoding can be seen as a hybrid between traditional tags (t) and monomorphic lightweight tags ($\widetilde{\mathsf{t?}}$): as in t, tags take the form of a function $\mathsf{t}\langle\sigma\rangle(\mathsf{t})$; as in $\widetilde{\mathsf{t?}}$, tags are omitted for types that are inferred monotonic.

The main novelty concerns the typing axioms. The $\widetilde{\mathsf{t?}}$ encoding omits all typing axioms for infinite types. In the polymorphic case, the infinite type σ might be an instance of a more general, potentially finite type for which tags are generated. For example, if α is tagged (because it is possibly non-monotonic) but its instance $\mathtt{list}(\alpha)$ is not (because it is infinite), there will be mismatches between tagged and untagged terms. Our solution is to add the typing axiom $\mathsf{t}\langle\mathtt{list}(\alpha)\rangle(\mathsf{Xs}) \approx \mathsf{Xs}$, which allows the prover to add or remove a tag for the infinite type $\mathtt{list}(\alpha)$. Such an axiom is sound for any monotonic type.

Definition 26 (Lightweight Tags t?). The *polymorphic lightweight type tags* encoding t? translates a polymorphic problem Φ over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \{\mathsf{t}^2\}, \mathcal{P}')$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for $\mathsf{a}^{\mathsf{ctor}}$. It is defined as $[\![\ ]\!]_{\mathsf{t?};\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$, where

$$[\![\mathsf{f}\langle\sigma\rangle(\bar{\mathsf{t}}\,)^\sigma]\!]_{\mathsf{t?}} = \lfloor \mathsf{f}\langle\sigma\rangle([\![\bar{\mathsf{t}}\,]\!]_{\mathsf{t?}})\rfloor \qquad [\![X^\sigma]\!]_{\mathsf{t?}} = \lfloor X \rfloor$$

with

$$\lfloor t^{\sigma} \rfloor = \begin{cases} t & \text{if } \sigma \rhd \Phi \\ t\langle\sigma\rangle(t) & \text{otherwise} \end{cases}$$

The encoding is complemented by the following typing axioms, where ρ is a type substitution and $\mathrm{TV}(\sigma\rho)$ denotes the type variables of $\sigma\rho$:

$$\forall \mathrm{TV}(\sigma\rho).\ \forall X : \sigma\rho.\ t\langle\sigma\rho\rangle(X) \approx X \quad \text{for } \sigma\rho \in \mathrm{Inf}(\Phi) \text{ such that } \sigma \not\rhd \Phi$$

Example 14. For the algebraic list problem of Example 2, $\mathsf{list}(\alpha)$ is monotonic by virtue of being infinite, whereas α and its instance b cannot be inferred monotonic. The t? encoding of the problem follows:

$$\forall A,\ Xs.\ \mathsf{t}(\mathsf{list}(A),\ Xs) \approx Xs$$

$$\begin{aligned}
&\forall A,\ X,\ Xs.\ \mathsf{nil} \not\approx \mathsf{cons}(A,\ \mathsf{t}(A,\ X),\ Xs)\\
&\forall A,\ Xs.\ Xs \approx \mathsf{nil} \vee (\exists Y,\ Ys.\ Xs \approx \mathsf{cons}(A,\ \mathsf{t}(A,\ Y),\ Ys))\\
&\forall A,\ X,\ Xs.\ \mathsf{t}(A,\ \mathsf{hd}(A,\ \mathsf{cons}(A,\ \mathsf{t}(A,\ X),\ Xs))) \approx \mathsf{t}(A,\ X) \wedge\\
&\qquad\qquad \mathsf{tl}(A,\ \mathsf{cons}(A,\ \mathsf{t}(A,\ X),\ Xs)) \approx Xs\\
&\exists X,\ Y,\ Xs,\ Ys.\ \mathsf{cons}(\mathsf{b},\ \mathsf{t}(\mathsf{b},\ X),\ Xs) \approx \mathsf{cons}(\mathsf{b},\ \mathsf{t}(\mathsf{b},\ Y),\ Ys) \wedge\\
&\qquad\qquad (\mathsf{t}(\mathsf{b},\ X) \not\approx \mathsf{t}(\mathsf{b},\ Y) \vee Xs \not\approx Ys)
\end{aligned}$$

The typing axiom allows any term to be typed as $\mathsf{list}(\alpha)$, which is sound because $\mathsf{list}(\alpha)$ is infinite. It would have been equally correct to provide separate axioms for nil, cons, and tl. Either way, the axioms are needed to remove the $\mathsf{t}(A, X)$ tags in case the proof requires reasoning about $\mathsf{list}(\mathsf{list}(\alpha))$.

The lighter encoding t?? protects only naked variables and introduces equations of the form $\mathsf{t}\langle\sigma\rangle(\mathsf{f}\langle\bar{\alpha}\rangle(\overline{X})) \approx \mathsf{f}\langle\bar{\alpha}\rangle(\overline{X})$ to add or remove tags around each function symbol f of a possibly non-monotonic type σ, and similarly for existential variables.

Definition 27 (Featherweight Tags t??)**.** The *polymorphic featherweight type tags* encoding t?? translates a polymorphic problem Φ over Σ into an untyped problem over Σ', where Σ' is as for t?. It is defined as $[\![\ ]\!]_{\mathsf{t??};\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$, where

$$[\![t_1 \approx t_2]\!]_{\mathsf{t??}} = \lfloor [\![t_1]\!]_{\mathsf{t??}} \rfloor \approx \lfloor [\![t_2]\!]_{\mathsf{t??}} \rfloor$$

$$[\![\exists X : \sigma.\ \varphi]\!]_{\mathsf{t??}} = \exists X : \sigma.\ \begin{cases} [\![\varphi]\!]_{\mathsf{t??}} & \text{if } \sigma \rhd \Phi \\ t\langle\sigma\rangle(X) \approx X \wedge [\![\varphi]\!]_{\mathsf{t??}} & \text{otherwise} \end{cases}$$

with

$$\lfloor t^{\sigma} \rfloor = \begin{cases} t & \text{if } \sigma \rhd \Phi \text{ or } t \text{ is not a universal variable} \\ t\langle\sigma\rangle(t) & \text{otherwise} \end{cases}$$

The encoding is complemented by typing axioms:

$$\forall \bar{\alpha}.\, \forall \bar{X} : \bar{\sigma}.\ \mathsf{t}\langle\sigma\rangle(\mathsf{f}\langle\bar{\alpha}\rangle(\bar{X})) \approx \mathsf{f}\langle\bar{\alpha}\rangle(\bar{X})$$
$$\text{for } \mathsf{f} : \forall \bar{\alpha}.\, \bar{\sigma} \to \sigma \in \mathcal{F} \text{ such that } \exists \rho.\, \sigma\rho \not\trianglerighteq \Phi$$
$$\forall \mathsf{TV}(\sigma\rho).\, \forall X : \sigma\rho.\ \mathsf{t}\langle\sigma\rho\rangle(X) \approx X$$
$$\text{for } \sigma\rho \in \mathsf{Inf}(\Phi) \text{ such that } \sigma \not\trianglerighteq \Phi$$
$$\forall \mathsf{TV}(\sigma).\, \exists X : \sigma.\ \mathsf{t}\langle\sigma\rangle(X) \approx X$$
$$\text{for } \sigma \not\trianglerighteq \Phi \text{ that is not an instance of the result type of } \mathsf{f} \in \mathcal{F} \text{ or a proper instance of } \tau \not\trianglerighteq \Phi$$

Example 15. The t?? encoding of Example 2 requires fewer tags than $\widetilde{\mathsf{t}}$?, at the cost of more type information:

$$\forall A, Xs.\ \mathsf{t}(A, \mathsf{hd}(A, Xs)) \approx \mathsf{hd}(A, Xs)$$
$$\forall A, Xs.\ \mathsf{t}(\mathsf{list}(A), Xs) \approx Xs$$
$$\forall A.\, \exists X.\ \mathsf{t}(A, X) \approx X$$

$$\forall A, X, Xs.\ \mathsf{nil} \not\approx \mathsf{cons}(A, X, Xs)$$
$$\forall A, Xs.$$
$$Xs \approx \mathsf{nil} \vee (\exists Y, Ys.\ \mathsf{t}(A, Y) \approx Y \wedge Xs \approx \mathsf{cons}(A, Y, Ys))$$
$$\forall A, X, Xs.$$
$$\mathsf{hd}(A, \mathsf{cons}(A, X, Xs)) \approx \mathsf{t}(A, X) \wedge$$
$$\mathsf{tl}(A, \mathsf{cons}(A, X, Xs)) \approx Xs$$
$$\exists X, Y, Xs, Ys.$$
$$\mathsf{t}(\mathsf{b}, X) \approx X \wedge \mathsf{t}(\mathsf{b}, Y) \approx Y \wedge$$
$$\mathsf{cons}(\mathsf{b}, X, Xs) \approx \mathsf{cons}(\mathsf{b}, Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)$$

Theorem 11 (Correctness of t?, t??). *The polymorphic type tags encodings* t? *and* t?? *are correct.*

Proof. It suffices to show the three conditions of Lemma 9. The proof is analogous to that of Theorem 7.

Mono: Both encodings tag any naked variables that occur in the original problem and that could obstruct the monotonicity calculus. The typing axioms $\mathsf{t}\langle\sigma\rangle(X) \approx X$ contain a naked variable, but they are inferred monotonic because σ must be infinite. Monotonicity follows by Theorem 10 and Lemma 6.

Sound: Given a model of Φ, we extend it to a model of $[\![\Phi]\!]_x$ by interpreting the type tag function $\mathsf{t}\langle\sigma\rangle$ as the identity.

COMPLETE: For t?, we use the same argument as in Theorem 7. The only difficulty is if Φ has a term of possibly non-monotonic type σ that in the ground proof is instantiated with an infinite type $\sigma\rho$: the corresponding instance of $[\![\Phi]\!]_{t??}$ will have an unnecessary type tag $t_{\sigma\rho}$. We fix this by applying the typing axiom $t_{\sigma\rho}(X) = X$ which we generate for infinite instances of possibly non-monotonic types.

For t??, we construct a canonical model of $[\![\Phi]\!]_{t??}$, from which we obtain a model of Φ by the converse construction to SOUND. The ground types σ occurring in Φ are partitioned into three classes:

(a) The possibly non-monotonic types ($\sigma \not\rhd \Phi$). The typing axioms $\exists X : \sigma.\ t\langle\sigma\rangle(X) \approx X$ and $t\langle\sigma\rangle(f\langle\bar{\alpha}\rangle(\bar{X})) \approx f\langle\bar{\alpha}\rangle(\bar{X})$ ensure that $t\langle\sigma\rangle$ is the identity for at least one element and for all well-typed terms; furthermore, the equations $t\langle\sigma\rangle(X) \approx X$ generated for existential variables ensures that $t\langle\sigma\rangle$ is the identity for witnesses, as required to apply Lemma 8.

(b) For the infinite instances of possibly non-monotonic types (types $\sigma\rho \in Inf(\Phi)$ such that $\sigma \not\rhd \Phi$), the typing axiom $t\langle\sigma\rangle(X) \approx X$ ensures that $t\langle\sigma\rangle$ is the identity.

(c) Values of the remaining monotonic types σ (such that $\sigma \rhd \Phi$) are never accessed through a tag function $t\langle\tau\rangle$—such a tag is generated only if $\tau \not\rhd \Phi$, but in that case σ must be infinite for $\sigma \rhd \Phi$ to be derivable.

The canonical model is obtained by removing the domain elements for which $t\langle\sigma\rangle$ is not the identity, appealing to Lemma 8. □

5.5 Monotonicity-Based Type Guards

Analogously to t?, the g? encoding is best understood as a hybrid between traditional guards (g) and monomorphic lightweight guards ($\widetilde{g}$?): as in g, guards take the form of a predicate $g\langle\sigma\rangle(t)$; as in $\widetilde{g}$?, guards are omitted for types that are inferred monotonic.

Once again, the main novelty concerns the typing axioms. The $\widetilde{g}$? encoding omits all typing axioms for infinite types. In the polymorphic case, the infinite type σ might be an instance of a more general, potentially finite type for which guards are generated. Our solution is to add the typing axiom $g\langle\sigma\rangle(X)$, which allows the prover to discharge any guard for the infinite type σ.

Definition 28 (Lightweight Guards g?)**.** The *polymorphic lightweight type guards* encoding g? translates a polymorphic problem Φ over Σ into an untyped problem over $\Sigma' = (\mathcal{F}', \mathcal{P}' \uplus \{g^2\})$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for a^{ctor}. It is defined as $[\![\ ]\!]_{g?;a^{ctor};e}$, where

$$[\![\forall X : \sigma.\ \varphi]\!]_{g?} = \forall X : \sigma. \begin{cases} [\![\varphi]\!]_{g?} & \text{if } \sigma \rhd \Phi \\ g\langle\sigma\rangle(X) \to [\![\varphi]\!]_{g?} & \text{otherwise} \end{cases}$$

$$[\![\exists X : \sigma.\ \varphi]\!]_{g?} = \exists X : \sigma. \begin{cases} [\![\varphi]\!]_{g?} & \text{if } \sigma \rhd \Phi \\ g\langle\sigma\rangle(X) \wedge [\![\varphi]\!]_{g?} & \text{otherwise} \end{cases}$$

The encoding is complemented by typing axioms:

$$\forall \bar{\alpha}.\ \forall \bar{X} : \bar{\sigma}.\ g\langle\sigma\rangle(f\langle\bar{\alpha}\rangle(\bar{X}))$$
$$\quad \text{for } f : \forall \bar{\alpha}.\ \bar{\sigma} \to \sigma \in \mathcal{F} \text{ such that } \exists \rho.\ \sigma\rho \not\rhd \Phi$$
$$\forall TV(\sigma\rho).\ \forall X : \bar{\sigma}\rho.\ g\langle\sigma\rho\rangle(X)$$
$$\quad \text{for } \sigma\rho \in Inf(\Phi) \text{ such that } \sigma \not\rhd \Phi$$
$$\forall TV(\sigma).\ \exists X : \sigma.\ g\langle\sigma\rangle(X)$$
$$\quad \text{for } \sigma \not\rhd \Phi \text{ that is not an instance of the result type of } f \in \mathcal{F} \text{ or a proper instance of } \tau \not\rhd \Phi$$

The featherweight cousin is a straightforward generalisation of g? along the lines of the generalisation of $\widetilde{g}?$ into $\widetilde{g}??$.

Definition 29 (Featherweight Guards g??)**.** The *polymorphic featherweight type guards* encoding g?? is identical to the lightweight encoding g? except that the condition "if $\sigma \rhd \Phi$" in the $\forall$ case is weakened to "if $\sigma \rhd \Phi$ or $X \notin NV(\varphi)$".

Example 16. The g?? encoding of Example 2 follows:

$$\forall A, Xs.\ g(A, hd(A, Xs)) \tag{1}$$
$$\forall A, Xs.\ g(list(A), Xs) \tag{2}$$
$$\forall A, X, Xs.\ nil \not\approx cons(A, X, Xs) \tag{3}$$
$$\forall A, Xs.\ Xs \approx nil \vee (\exists Y, Ys.\ g(A, Y) \wedge Xs \approx cons(A, Y, Ys)) \tag{4}$$
$$\forall A, X, Xs.\ g(A, X) \to \tag{5}$$
$$\quad hd(A, cons(A, X, Xs)) \approx X \wedge tl(A, cons(A, X, Xs)) \approx Xs \tag{6}$$
$$\exists X, Y, Xs, Ys.\ g(b, X) \wedge g(b, Y) \wedge \tag{7}$$
$$\quad cons(b, X, Xs) \approx cons(b, Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys) \tag{8}$$

Theorem 12 (Correctness of g?, g??). *The polymorphic type guards encodings* g? *and* g?? *are correct.*

Proof. It suffices to show the three conditions of Lemma 9. The proof is analogous to that of Theorem 8.

MONO: Both encodings guard any naked variables that occur in the original problem and could prevent the problem from being monotonic. Given a model $\mathcal{M}$ of $[\![\Phi]\!]_x$ where each σ is interpreted as D_σ, we construct a model $\mathcal{M}'$ of $[\![\Phi]\!]_x$ where σ is interpreted as $D_\sigma \uplus D'_\sigma$. We choose a representative $a \in D_\sigma$ and extend the interpretations of all symbols so that they coincide on a and each $a' \in D'_\sigma$ except for $g\langle\sigma\rangle(a')$, which is set to true if σ is an instance of a type in $\mathrm{Inf}(\Phi)$ and false otherwise. This is compatible with the typing axioms (including $g\langle\sigma\rangle(X)$ for infinite σ) and effectively prevents a' from arising naked if σ is not infinite.

SOUND: Given a model of Φ, we extend it to a model of $[\![\Phi]\!]_x$ by interpreting the type guard $g\langle\sigma\rangle$ as true everywhere.

COMPLETE: The ground types σ occurring in Φ are partitioned into three classes, as in the proof of Theorem 11. The canonical model is obtained by removing the domain elements for which $g\langle\sigma\rangle$ is false, appealing to Lemma 8. □

6 Alternative, Cover-Based Encoding of Polymorphism

An issue with t?, t??, g?, and g?? is that they clutter the generated problem with type arguments. In that respect, the traditional t and g encodings are superior—t omits all non-phantom type arguments, and g omits all inferrable type arguments. This would be unsound for the monotonicity-based encodings, because these leave out many of the protectors that implicitly "carry", or "cover", the type arguments in the traditional encodings. Nonetheless, an alternative is possible: by keeping more protectors around, we can omit inferrable type arguments. Let us first rigorously define this notion of term arguments "covering" type arguments.

Definition 30 (Cover). Let $s : \forall\bar{\alpha}.\ \bar{\sigma} \to \sigma \in \mathcal{F} \uplus \mathcal{P}$. A (*type argument*) *cover* $C \subseteq \{1, \ldots, |\bar{\sigma}|\}$ for s is a set of term argument indices such that any inferrable type argument can be inferred from a term argument whose index is in C. A cover C of s is *minimal* if no proper subset of C

is a cover for s. We let Cover_s denote an arbitrary but fixed minimal cover of s and extend the notion to the distinguished symbols t, g by taking $\mathit{Cover}_t = \mathit{Cover}_g = \emptyset$ (cf. Definition 8).

For example, $\{1\}$ and $\{2\}$ are minimal covers for $\mathsf{cons} : \forall\alpha.\ \alpha \times \mathsf{list}(\alpha) \to \mathsf{list}(\alpha)$, and $\{1, 2\}$ is also a cover. As canonical cover, we arbitrarily choose $\mathit{Cover}_{\mathsf{cons}} = \{1\}$.

The cover-based encodings t@ and g@ introduced below ensure that each term argument position that is part of its enclosing function or predicate's cover has a unique type associated with it, from which the omitted type arguments can be inferred. For example, t@ translates the term $\mathsf{cons}\langle\alpha\rangle(X, Xs)$ to $\mathsf{cons}(\mathsf{t}(A, X), Xs)$ with a type tag around X, effectively preventing an ill-typed instantiation of X that would result in the wrong type argument being inferred. But we do not need to protect the second argument, Xs—it is sufficient to introduce enough protectors to "cover" all the inferrable type arguments. We call variables that occur in their enclosing symbol's cover (and hence that "carry" some type arguments) "undercover variables".

Definition 31 (Undercover Variable)**.** The set of *undercover variables* $\mathit{UV}(\varphi)$ of a formula φ is defined by the equations

$$\mathit{UV}(t_1 \approx t_2) = (\{t_1, t_2\} \cap \mathcal{V}) \cup \mathit{UV}(t_1, t_2)$$

$$\begin{aligned}
\mathit{UV}(f\langle\bar\sigma\rangle(\bar t)) &= \lfloor \bar t \rfloor_f \cup \mathit{UV}(\bar t) & \mathit{UV}(X) &= \emptyset \\
\mathit{UV}(p\langle\bar\sigma\rangle(\bar t)) &= \lfloor \bar t \rfloor_p \cup \mathit{UV}(\bar t) & \mathit{UV}(\forall\alpha.\ \varphi) &= \mathit{UV}(\varphi) \\
\mathit{UV}(\neg\, p\langle\bar\sigma\rangle(\bar t)) &= \lfloor \bar t \rfloor_p \cup \mathit{UV}(\bar t) & \mathit{UV}(t_1 \not\approx t_2) &= \mathit{UV}(t_1, t_2) \\
\mathit{UV}(\varphi_1 \wedge \varphi_2) &= \mathit{UV}(\varphi_1, \varphi_2) & \mathit{UV}(\forall X : \sigma.\ \varphi) &= \mathit{UV}(\varphi) \\
\mathit{UV}(\varphi_1 \vee \varphi_2) &= \mathit{UV}(\varphi_1, \varphi_2) & \mathit{UV}(\exists X : \sigma.\ \varphi) &= \mathit{UV}(\varphi) - \{X\}
\end{aligned}$$

where $\lfloor \bar t \rfloor_s = \{t_j \mid j \in \mathit{Cover}_s\} \cap \mathcal{V}$ and $\mathit{UV}(\bar t) = \bigcup_j \mathit{UV}(t_j)$.

Undercover variables are reminiscent of naked variables. This is no coincidence: naked variables effectively carry the implicit type argument of $\approx : \forall\alpha.\ \alpha \times \alpha \to o$, and undercover variables generalise this to arbitrary predicate and function symbols. Equality is special in two respects, though: its only sound cover is effectively $\{1, 2\}$ because of the equality axioms (which are built into the provers, without any protectors), and the negative case is optimised to take advantage of disequality's fixed semantics.

6.1 Cover-Based Type Tags

The cover-based encoding t@ is similar to the traditional encoding t, except that it tags only undercover occurrences of variables and requires typing axioms to add or remove tags around function terms.

Definition 32 (Cover Tags t@)**.** The *polymorphic cover-based type tags* encoding t@ translates a polymorphic problem over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \mathcal{K} \uplus \{\mathsf{t}^2\}, \mathcal{P}')$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for $\mathsf{a}^{\mathsf{ninf}}$. It is defined as $[\![\ ]\!]_{\mathsf{t@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}$, where

$$\begin{aligned}
[\![\mathsf{f}\langle\bar{\sigma}\rangle(\bar{t}\,)]\!]_{\mathsf{t@}} &= \mathsf{f}\langle\bar{\sigma}\rangle(\lfloor[\![\bar{t}\,]\!]_{\mathsf{t@}}\rfloor_{\mathsf{f}}) \\
[\![\mathsf{p}\langle\bar{\sigma}\rangle(\bar{t}\,)]\!]_{\mathsf{t@}} &= \mathsf{p}\langle\bar{\sigma}\rangle(\lfloor[\![\bar{t}\,]\!]_{\mathsf{t@}}\rfloor_{\mathsf{p}}) \\
[\![\neg\, \mathsf{p}\langle\bar{\sigma}\rangle(\bar{t}\,)]\!]_{\mathsf{t@}} &= \neg\, \mathsf{p}\langle\bar{\sigma}\rangle(\lfloor[\![\bar{t}\,]\!]_{\mathsf{t@}}\rfloor_{\mathsf{p}}) \\
[\![t_1 \approx t_2]\!]_{\mathsf{t@}} &= \lfloor[\![t_1]\!]_{\mathsf{t@}}\rfloor_{\approx} \approx \lfloor[\![t_2]\!]_{\mathsf{t@}}\rfloor_{\approx} \\
[\![\exists X : \sigma.\ \varphi]\!]_{\mathsf{t@}} &= \exists X : \sigma.\ \mathsf{t}\langle\sigma\rangle(X) \approx X \wedge [\![\varphi]\!]_{\mathsf{t@}}
\end{aligned}$$

The auxiliary function $\lfloor(t_1^{\sigma_1}, \ldots, t_n^{\sigma_n})\rfloor_s$ returns a vector $(u_1, \ldots, u_n)$ such that

$$u_j = \begin{cases} t_j & \text{if } j \notin \mathit{Cover}_s \text{ or } t_j \text{ is not a universal variable} \\ \mathsf{t}\langle\sigma_j\rangle(t_j) & \text{otherwise} \end{cases}$$

taking $\mathit{Cover}_{\approx} = \{1, 2\}$. The encoding is complemented by typing axioms:

$$\forall\bar{\alpha}.\ \forall\bar{X} : \bar{\sigma}.\ \mathsf{t}\langle\sigma\rangle(\mathsf{f}\langle\bar{\alpha}\rangle(\lfloor\bar{X}\rfloor_{\mathsf{f}})) \approx \mathsf{f}\langle\bar{\alpha}\rangle(\lfloor\bar{X}\rfloor_{\mathsf{f}}) \qquad \text{for } \mathsf{f} : \forall\bar{\alpha}.\ \bar{\sigma} \to \sigma \in \mathcal{F}$$
$$\forall\alpha.\ \exists X : \alpha.\ \mathsf{t}\langle\alpha\rangle(X) \approx X$$

Example 17. The t@ encoding of Example 2 is as follows:

$$\forall A.\ \mathsf{t}(\mathsf{list}(A), \mathsf{nil}(A)) \approx \mathsf{nil}(A)$$
$$\forall A, X, Xs.\ \mathsf{t}(\mathsf{list}(A), \mathsf{cons}(\mathsf{t}(A, X), Xs)) \approx \mathsf{cons}(\mathsf{t}(A, X), Xs)$$
$$\forall A, Xs.\ \mathsf{t}(\mathsf{list}(A), \mathsf{hd}(\mathsf{t}(\mathsf{list}(A), Xs))) \approx \mathsf{hd}(\mathsf{t}(\mathsf{list}(A), Xs))$$
$$\forall A, Xs.\ \mathsf{t}(A, \mathsf{tl}(\mathsf{t}(\mathsf{list}(A), Xs))) \approx \mathsf{tl}(\mathsf{t}(\mathsf{list}(A), Xs))$$

$$\forall A, X, Xs.\ \mathsf{nil}(A) \not\approx \mathsf{cons}(\mathsf{t}(A, X), Xs)$$

$$
\begin{aligned}
&\forall A,\, Xs.\ \mathsf{t}(\mathsf{list}(A),\, Xs) \approx \mathsf{nil}(A) \vee \\
&\quad (\exists Y,\, Ys.\ \mathsf{t}(A,\, Y) \approx Y \wedge \\
&\quad \mathsf{t}(\mathsf{list}(A),\, Ys) \approx Ys \wedge \mathsf{t}(\mathsf{list}(A),\, Xs) \approx \mathsf{cons}(Y,\, Ys)) \\
&\forall A,\, X,\, Xs. \\
&\quad \mathsf{hd}(\mathsf{cons}(\mathsf{t}(A,\, X),\, Xs)) \approx \mathsf{t}(A,\, X) \wedge \\
&\quad \mathsf{tl}(\mathsf{cons}(\mathsf{t}(A,\, X),\, Xs)) \approx \mathsf{t}(\mathsf{list}(A),\, Xs) \\
&\exists X,\, Y,\, Xs,\, Ys. \\
&\quad \mathsf{t}(\mathsf{b},\, X) \approx X \wedge \mathsf{t}(\mathsf{b},\, Y) \approx Y \wedge \\
&\quad \mathsf{t}(\mathsf{list}(\mathsf{b}),\, Xs) \approx Xs \wedge \mathsf{t}(\mathsf{list}(\mathsf{b}),\, Ys) \approx Ys \wedge \\
&\quad \mathsf{cons}(X,\, Xs) \approx \mathsf{cons}(Y,\, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)
\end{aligned}
$$

In Section 5, we showed that it is sound to omit non-inferrable type arguments to constructors for monotonicity-based encodings, yielding nil instead of $\mathsf{nil}(A)$ in the translation. For cover-based encodings, this would be unsound, as shown by the following counterexample: $\mathsf{q}\langle\mathsf{a}\rangle(\mathsf{nil}\langle\mathsf{a}\rangle) \wedge \neg\, \mathsf{q}\langle\mathsf{b}\rangle(\mathsf{nil}\langle\mathsf{b}\rangle)$ is satisfiable, but the naive translation to $\mathsf{q}(\mathsf{nil}) \wedge \neg\, \mathsf{q}(\mathsf{nil})$ is unsatisfiable. The counterexample does not apply to monotonicity-based encodings, because q would be passed an encoded type argument: $\mathsf{q}(\mathsf{a},\, \mathsf{nil}) \wedge \neg\, \mathsf{q}(\mathsf{b},\, \mathsf{nil})$ is satisfiable.

Theorem 13 (Correctness of t@). *The cover-based type tags encoding* t@ *is correct.*

Proof. Sound: Given a model of Φ, we extend it to a model $\mathcal{M}$ of $\Phi' = [\![\Phi]\!]_{\mathsf{t@}}$ by interpreting $\mathsf{t}\langle\sigma\rangle$ as the identity. We assume the domains of $\mathcal{M}$ are disjoint and choose from each domain a representative element. We then construct a model $\mathcal{M}'$ of $[\![\Phi']\!]_{\mathsf{a}^{\mathsf{phan}};\mathsf{e}}$ from $\mathcal{M}$ as follows. The domain for $\mathcal{M}'$ is the disjoint union of the domains for $\mathcal{M}$ and of the term universe of encoded ground types over $\mathcal{K}$. For each symbol $\mathsf{s} : \forall\bar{\alpha}.\ \bar{\sigma} \to \sigma \in \mathcal{F} \uplus \mathcal{P} \uplus \{\mathsf{t}\}$, the entry for $\mathsf{s}(\langle\!\langle\bar{\tau}\rangle\!\rangle,\, \bar{a})$ in $\mathcal{M}'$ is set as follows:

1. If $\mathsf{s}\langle\bar{\sigma}\rangle(\bar{a})$ is defined in $\mathcal{M}$ (i.e. the $\bar{a}$'s lie within the interpretations of their respective domains) for some $\bar{\sigma}$ such that $\mathsf{phan}_{\mathsf{s}}(\bar{\sigma}) = \bar{\tau}$, set the entry for $\mathsf{s}(\langle\!\langle\bar{\tau}\rangle\!\rangle,\, \bar{a})$ to the interpretation of $\mathsf{s}\langle\bar{\sigma}\rangle(\bar{a})$ by $\mathcal{M}$. (The domain disjointness assumption, together with the condition $\mathsf{phan}_{\mathsf{s}}(\bar{\sigma}) = \bar{\tau}$, ensures that there exists at most one vector $\bar{\sigma}$ such that $\mathsf{s}\langle\bar{\sigma}\rangle(\bar{a})$ is defined.)

2. Otherwise, attempt to repair the vector $\bar{a}$ by replacing any a_j that is not in its domain by a "well-typed" representative, while keep-

ing all a_j's such that $j \in Cover_s$ unchanged. If this construction succeeds, proceed as in step 1.

3. Otherwise, choose some arbitrary domain element (if s is a function) or truth value (if s is a predicate).

We must show that $\mathcal{M}'$ is a model of $[\![\Phi']\!]_{a^{phan};e}$. Thanks to step 1, the semantics of the problems' formulas (including the typing axioms) coincide if universal variables are instantiated by "well-typed" values. If all tagged variables in a formula are instantiated by "well-typed" values but some of the untagged variables are not, the semantics of the formula coincides with, or is weaker than (cf. the proof of Theorem 7), that of an already considered "well-typed" instance thanks to step 2, since "ill-typed" values are interpreted the same way as the representative element. Finally, if some of the tagged variables are instantiated by "ill-typed" values, the tags return the representatives, and non-tagged instances of the same variables are treated as the representatives, so again this case coincides with an already considered case.

Complete: The a^{phan} encoding is complete by Theorem 3. It remains to show that $[\![\ ]\!]_{t@}$ is complete. Given a model of $[\![\Phi]\!]_{t@}$, we construct a canonical model of $[\![\Phi]\!]_{t@}$, from which we extract a model of Φ, as in the proof of Theorem 11. The typing axioms ensure that $t\langle\sigma\rangle$ is the identity for at least one element and for all "well-typed" terms; furthermore, the equations $t\langle\sigma\rangle(X) \approx X$ generated for existential variables ensures that $t\langle\sigma\rangle$ is the identity for witnesses, as required to apply Lemma 8. □

6.2 Cover-Based Type Guards

The cover-based g@ encoding is defined analogously.

Definition 33 (Cover Guards g@). The *polymorphic cover-based type guards* encoding g@ is identical to the traditional g encoding except for the $\forall$ case:

$$[\![\forall X : \sigma.\ \varphi]\!]_{g@} = \forall X : \sigma. \begin{cases} [\![\varphi]\!]_{g@} & \text{if } X \notin \mathcal{U}V(\varphi) \\ g\langle\sigma\rangle(X) \to [\![\varphi]\!]_{g@} & \text{otherwise} \end{cases}$$

The encoding is complemented by typing axioms:

$$\forall \bar{\alpha}.\ \bar{X} : \bar{\sigma}.\ \left(\bigwedge_{j \in Cover_f} g\langle\sigma_j\rangle(X_j)\right) \to g\langle\sigma\rangle(f\langle\bar{\alpha}\rangle(\bar{X}))$$
$$\text{for } f : \forall \bar{\alpha}.\ \bar{\sigma} \to \sigma \in \mathcal{F}$$
$$\forall \alpha.\ \exists X : \alpha.\ g\langle\alpha\rangle(X)$$

Example 18. The g@ encoding of the algebraic list problem is identical to the g encoding (Example 7), except that the guard $\mathsf{g}(\mathsf{list}(A), Xs)$ is omitted in two of the axioms:

$$\forall A, X, Xs.\ \mathsf{g}(A, X) \to \mathsf{g}(\mathsf{list}(A), \mathsf{cons}(X, Xs))$$
$$\forall A, X, Xs.\ \mathsf{g}(A, X) \to \mathsf{nil}(A) \not\approx \mathsf{cons}(X, Xs)$$

Theorem 14 (Correctness of g@). *The cover-based type guards encoding* g@ *is correct.*

Proof. Sound: Given a model of Φ, we extend it to a model $\mathcal{M}$ of $\Phi' = [\![\Phi]\!]_{\mathsf{g@}}$ by interpreting $\mathsf{g}\langle\sigma\rangle$ as the true predicate. We assume the domains of $\mathcal{M}$ are disjoint and choose from each domain a representative element. We then construct a model $\mathcal{M}'$ exactly as in the proof of Theorem 13, except that $\mathsf{g}(\langle\!\langle\sigma\rangle\!\rangle, a)$ is set to be false for "ill-typed" values a (i.e. $a \notin [\![\sigma]\!]^{\mathcal{M}}$). We must show that $\mathcal{M}'$ is a model of $[\![\Phi']\!]_{\mathsf{a}^{\mathsf{phan}};\mathsf{e}}$. Thanks to step 1 of the construction of $\mathcal{M}'$, the semantics coincide if universal variables are instantiated by "well-typed" values. If all guarded variables are instantiated by "well-typed" values but some of the unguarded variables are not, the semantics of the formula coincides with, or is weaker than, that of an already considered "well-typed" instance thanks to step 2. Finally, if some of the guarded variables are instantiated by "ill-typed" values, the guard returns false, making the subformula true.

Complete: The proof is analogous to the corresponding case of Theorems 11. □

7 Implementation

Our research on polymorphic type encodings was driven by Sledgehammer, a component of Isabelle/HOL that harnesses first-order automatic theorem provers to discharge interactive proof obligations. The tool heuristically selects hundreds of background facts, translates them to untyped or monomorphic first-order logic, invokes the external provers in parallel, and reconstructs machine-generated proofs in Isabelle.

All the encodings presented in this paper, including the traditional encodings, are implemented in Sledgehammer and can be used to target external first-order provers. The rest of this section considers implementation issues in more detail.

7.1 Finite Monomorphisation Algorithm

The monomorphisation algorithm implemented in Sledgehammer translates a polymorphic problem into a monomorphic problem by heuristically instantiating type variables. It involves three stages:

1. Separate the monomorphic and the polymorphic formulas, and collect all symbols occurring in the monomorphic formulas (the "mono-symbols").

2. For each polymorphic axiom, stepwise refine a set of substitutions, starting from the singleton set containing only the empty substitution, by matching known mono-symbols against their polymorphic counterparts in the axiom. So long as new mono-symbols emerge, collect them and repeat this stage.

3. Apply the substitutions to the corresponding polymorphic formulas. Only keep fully monomorphic formulas.

To ensure termination, the iterations performed in stage 2 are limited to a configurable number K. To curb the exponential growth, the algorithm also enforce an upper bound Δ on the number of new formulas. Sledgehammer operates with $\mathsf{K} = 3$ and $\Delta = 200$ by default, so that a problem with 500 formulas comprises at most 700 formulas after monomorphisation. Experiments found these values suitable. Given formulas about $\mathtt{b}$ and $\mathtt{list}(\alpha)$, the third iteration already generates $\mathtt{list}(\mathtt{list}(\mathtt{list}(\mathtt{b})))$ instances; adding yet another layer of $\mathtt{list}$ is unlikely to help. Increasing Δ sometimes helps solve more problems, but its potential for clutter is real.

7.2 Extension to Higher-Order Logic

Sledgehammer's source logic, polymorphic higher-order logic (HOL) with axiomatic type classes [Wenzel, 1997], is not the same as the polymorphic first-order logic considered in this paper. The translation is a three-step process, where the first and last step may be omitted, depending on whether monomorphisation is desired and whether the target prover supports monomorphic types:

1. *Optionally monomorphise the problem.*

2. *Eliminate the higher-order constructs* [Meng and Paulson, 2008, §2.1]. λ-abstractions are rewritten to SK combinators or to supercombinators (λ-lifting). Functions are passed varying numbers of

arguments via an apply operator $\mathsf{hAPP} : \forall\alpha\,\beta.\ \mathtt{fun}(\alpha, \beta) \times \alpha \to \beta$ (where $\mathtt{fun}$ is uninterpreted). Boolean terms are converted to formulas using a unary predicate $\mathsf{hBOOL} : \mathtt{bool} \to o$ (where $\mathtt{bool}$ is uninterpreted).

3. *Encode the type information.* Polymorphic types are encoded using the techniques described in this paper. Type classes are essentially sets of types; they are encoded as polymorphic predicates $\forall\alpha.\ o$, where α is a phantom type variable. For example, a predicate $\mathsf{linorder} : \forall\alpha.\ o$ could be used to restrict the axioms specifying that $\mathsf{less} : \forall\alpha.\ \alpha \times \alpha \to o$ is a linear order to those types that satisfy the $\mathsf{linorder}$ predicate. The type class hierarchy is expressible as Horn clauses [Meng and Paulson, 2008, §2.3].

The symbol hAPP would hugely burden problems if it were introduced systematically for all arguments to functions. To reduce clutter, Sledgehammer computes the minimum arity n needed for each symbol and pass the first n arguments directly, falling back on hAPP for additional arguments. In general, more arguments can be passed directly if monomorphisation is performed before hAPP is introduced, because each monomorphic instance of a polymorphic symbol is considered individually. Similar observations can be made for hBOOL.

7.3 Infinite Types and Constructors

The monotonicity calculus $\rhd$ is parameterised by a set $\mathrm{Inf}(\Phi)$ of infinite types. One could employ an approach similar to that implemented in Infinox [Claessen and Lilliestrom, 2011] to automatically infer finite unsatisfiability of types. This tool relies on various proof principles to show that a set of untyped first-order formulas only has models with infinite domains. For example, it can infer that $\mathtt{list_b}$ is infinite in Example 3 because $\mathsf{cons}_\mathtt{b}$ is injective in its second argument but not surjective. However, in a proof assistant such as Isabelle, it is simpler to exploit metainformation available through introspection. Isabelle's datatypes are registered with their constructors; if some of them are recursive, or take an argument of an infinite type, the datatype must be infinite and hence monotonic.

More specifically, the monotonicity inference is run on the entire problem and maintains two finite sets of polymorphic types: the surely infinite types J and the possibly non-monotonic types N. Every type of a naked variable in the problem is tested for infinity. If the test succeeds, the type is inserted into J; otherwise, it is inserted into N. Simplifications

are performed: there no need to insert σ to J or N if it is an instance of a type already in the set; when inserting σ to a set, it is safe to remove any type in the set that is an instance of σ. The monotonicity check then becomes

$$\sigma \rhd \Phi \iff (\exists \tau \in \mathrm{J}.\, \exists \rho.\, \sigma = \tau\rho) \vee (\forall \tau \in \mathrm{N}.\, \nexists \rho.\, \sigma\rho = \tau\rho)$$

Introspection also plays a role in the $\mathsf{a}^{\mathsf{ctor}}$ encoding and its derivatives (t?, t??, g?, and g??). These are parameterised by a set of constructors $\mathrm{Ctor}(\Phi)$. Querying the datatype package is again simpler than attempting to detect distinctness and injectivity axioms in individual problems.

7.4 Proof Reconstruction

To guard against bugs in the external provers, Sledgehammer reconstructs machine-generated proofs in Isabelle. This is usually accomplished by the *metis* proof method [Paulson and Susanto, 2007], supplying it with the short list of facts referenced in the proof found by the prover. The proof method is based on the Metis prover [Hurd, 2003], a complete resolution prover for untyped first-order logic. The *metis* call is all that remains from the Sledgehammer invocation in the Isabelle theory, which can then be replayed without external provers. Given only a handful of facts, *metis* usually succeeds within milliseconds. Prior to our work, a large share of the reconstruction failures were caused by type-unsound proofs found by the external provers, due to the use of the unsound encoding a [Böhme and Nipkow, 2010, §4.1]. We now replaced the internals of Sledgehammer and *metis* so that they use a translation module supporting all the type encodings described in this paper. However, despite the typing information, individual inferences in Metis can be ill-typed when types are reintroduced, causing the *metis* proof method to fail. There are three main failure scenarios.

First, the prover may in principle instantiate variables with "ill-typed" terms at any point in the proof. Fortunately, this hardly ever arises in practice, because like other resolution provers Metis heavily restricts paramodulation from and into variables [Bachmair et al., 1995].

An issue that is more likely to plague users concerns the infinite types $\mathrm{Inf}(\Phi)$. Assuming *nat* is known to be infinite, the monotonicity-based encodings will not introduce any protectors around the naked variables M and N when translating the problem

$$\mathsf{on} \not\approx \mathsf{off}^{\mathsf{state}} \wedge (\forall \mathrm{X}, \mathrm{Y} : \mathrm{nat}.\ \mathrm{X} \approx \mathrm{Y})$$

(presumably a negated conjecture). That problem is satisfiable on its own but unsatisfiable with respect to the background theory. Untyped provers will happily instantiate X and Y with on and off to derive a contradiction; and no "type-sound" proof is possible unless we also provide characteristic theorems for *nat*. In general, we would need to provide infinity axioms for all types in $\mathrm{Inf}(\Phi)$ to make the encoding sound irrespective of the background theory; for example:

$$\forall \mathsf{N} : \mathit{nat}.\ \mathsf{zero} \not\approx \mathsf{suc}(\mathsf{N}) \quad \forall \mathsf{M}, \mathsf{N} : \mathit{nat}.\ \mathsf{suc}(\mathsf{M}) \approx \mathsf{suc}(\mathsf{N}) \to \mathsf{M} \approx \mathsf{N}$$

Although this now makes a sound proof possible (by instantiating X and Y with zero and suc(zero)), it does not prevent the prover from discovering the spurious proof with on and off, which cannot be reconstructed by *metis*.

A similar issue affects the constructors $\mathrm{Ctor}(\Phi)$. Let $\mathsf{c}, \mathsf{d} : \forall \alpha.\ \mathsf{k}(\alpha)$ be among the constructors for k. The encodings based on $\mathsf{a}^{\mathsf{ctor}}$ will translate the satisfiable problem

$$\mathsf{c}\langle \mathsf{a} \rangle \not\approx \mathsf{d}\langle \mathsf{a} \rangle \wedge \mathsf{c}\langle \mathsf{b} \rangle \approx \mathsf{d}\langle \mathsf{b} \rangle$$

into an unsatisfiable one: $\mathsf{c} \not\approx \mathsf{d} \wedge \mathsf{c} \approx \mathsf{d}$. In general, we would need to provide distinctness and injectivity axioms for all constructors in $\mathrm{Ctor}(\Phi)$ to make the encoding sound irrespective of the background theory; for example: $\forall \alpha.\ \mathsf{c}\langle \alpha \rangle \not\approx \mathsf{d}\langle \alpha \rangle$.

We stress that the issues above do not indicate unsoundness in our encodings. Rather, we use metainformation from Isabelle to guide the translation; the translation is then sound assuming that the metainformation holds. If we do not encode that metainformation in the typed problem, it might have a model where the metainformation is false, while the translation might exploit the metainformation and be unsatisfiable, as in the examples above. Proof reconstruction will then fail.

Although the above scenarios rarely occur in practice, it would be more satisfactory if proof reconstruction were always possible. A solution would be to connect our formalised soundness proofs with a verified checker for untyped first-order proofs. This remains for future work.

8 Evaluation

To evaluate the type encodings described in this paper, we put together two sets of 1000 polymorphic first-order problems originating from

10 existing Isabelle theories, translated with Sledgehammer's help (100 problems per theory).[2] Nine of the theories are the same as in a previous evaluation [Blanchette et al., 2011]; the tenth one is an optimality proof for Huffman's algorithm. Our test data are publicly available [Blanchette et al., 2012].

The problems in the first benchmark set include about 50 heuristically selected facts (before monomorphisation); that number is increased to 500 for the second set, to reveal how well the encodings scale with the problem size.

We evaluated each type encoding with five modern automatic theorem provers: the resolution provers E 1.6 [Schulz, 2004], SPASS 3.8ds [Weidenbach, 2001], and Vampire 2.6 [Riazanov and Voronkov, 2002], the instantiation prover iProver 0.99 [Korovin, 2009], and the SMT solver Z3 4.0 [de Moura and Bjørner, 2008]. To make the evaluation more informative, we also tested the provers' native support for monomorphic types where it is available; it is referred to as $\widetilde{\mathsf{n}}$. Each prover was invoked with the set of options we had previously determined worked best for Sledgehammer.[3] The provers were granted 20 seconds of CPU time per problem on one core of a 3.06 GHz Dual-Core Intel Xeon processor. Most proofs were found within a few seconds; a higher time limit would have had little impact on the success rate [Böhme and Nipkow, 2010, §4]. To avoid giving the unsound encodings (e, a, and $\mathsf{a}^{\mathsf{ctor}}$) an unfair advantage, for these proof search was followed by a certification phase that attempted to re-find the proof using a combination of sound encodings, based on its referenced facts. This phase slightly penalises the unsound encodings by rejecting a few sound proofs, but such is the price of unsoundness.

Figures 4.2 and 4.3 give, for each combination of prover and encoding, the number of solved problems from each problem set. Rows marked with ~ concern the monomorphic encodings. The encodings $\widetilde{\mathsf{a}}$, $\widetilde{\mathsf{a}}^{\mathsf{ctor}}$, $\widetilde{\mathsf{t}}$@, and $\widetilde{\mathsf{g}}$@ are omitted; the first two coincide with $\widetilde{\mathsf{e}}$, whereas $\widetilde{\mathsf{t}}$@ and $\widetilde{\mathsf{g}}$@ are identical to versions of $\widetilde{\mathsf{t}}$?? and $\widetilde{\mathsf{g}}$?? that treat all types as possibly non-monotonic. We observe the following:

- Among the encodings to untyped first-order logic, the monomor-

[2] The TPTP benchmark suite [Sutcliffe, 2009], which is customarily used for evaluating theorem provers, has just begun collecting polymorphic (TFF1) problems [Blanchette and Paskevich, 2012, §6].

[3] The setup for E was suggested by Stephan Schulz and includes the little known "symbol offset" weight function. We ran iProver with the default setup, SPASS in Isabelle mode, Vampire in CASC mode, and Z3 in TPTP mode with model-based quantifier instantiation enabled.

		e	a	a^{ctor}	t	t?	t??	t@
E		319	**328**	**328**	304	298	319	258
	~	333	–	–	325	337	333	–
iProver		243	220	225	**249**	190	236	157
	~	233	–	–	263	261	**264**	–
SPASS		276	286	286	267	272	**296**	196
	~	298	–	–	289	304	**310**	–
Vampire		**325**	312	312	319	307	314	260
	~	330	–	–	327	336	338	–
Z3		299	**321**	**321**	288	236	304	290
	~	319	–	–	300	313	319	–

		g	g?	g??	g@	n
E		267	316	323	285	–
	~	320	336	**340**	–	–
iProver		173	235	243	219	–
	~	222	259	261	–	–
SPASS		237	283	294	262	–
	~	291	308	305	–	309
Vampire		255	289	300	270	–
	~	299	337	**340**	–	332
Z3		263	281	301	275	–
	~	322	318	321	–	**325**

Figure 4.2: Number of solved problems with 50 facts

		e	a	a^{ctor}	t	t?	t??	t@
E		116	**361**	360	263	275	347	228
	~	393	–	–	328	390	397	–
iProver		243	212	207	231	202	**262**	135
	~	210	–	–	243	246	245	–
SPASS		131	292	292	262	245	**299**	164
	~	331	–	–	293	326	330	–
Vampire		120	**341**	**341**	277	281	314	212
	~	393	–	–	309	379	382	–
Z3		281	355	**357**	250	238	350	279
	~	354	–	–	268	343	346	–

		g	g?	g??	g@	n
E		216	344	349	262	–
	~	337	393	**401**	–	–
iProver		140	242	257	169	–
	~	180	**247**	241	–	–
SPASS		164	283	296	208	–
	~	237	320	334	–	**356**
Vampire		171	271	299	241	–
	~	265	390	**403**	–	372
Z3		213	291	351	268	–
	~	328	**355**	349	–	350

Figure 4.3: Number of solved problems with 500 facts

		e	a	a^{ctor}	t	t?	t??	t@
Clauses		832	901	901	918	947	1129	1307
	~	1188	–	–	1207	1207	1273	–
Literals per clause		2.55	2.77	2.77	2.77	2.71	2.97	3.33
	~	2.58	–	–	2.58	2.58	2.57	–
Symbols per atom		6.5	8.5	8.5	21.0	18.4	11.8	9.9
	~	6.9	–	–	11.7	7.7	6.9	–
Symbols ('000)		13.8	21.2	21.1	53.4	47.4	39.7	43.1
	~	21.0	–	–	36.4	24.0	22.5	–

		g	g?	g??	g@
Clauses		1304	1127	1126	1304
	~	1890	1273	1273	–
Literals per clause		5.88	4.63	3.20	5.01
	~	5.41	2.87	2.66	–
Symbols per atom		5.8	8.6	10.7	5.9
	~	4.3	6.2	6.6	–
Symbols ('000)		44.8	45.0	38.5	38.5
	~	44.5	22.8	22.3	–

Figure 4.4: Average size of clausified problems with 500 facts

phic monotonicity-based encodings (especially $\widetilde{\mathsf{g}}$?? but also $\widetilde{\mathsf{t}}$??, $\widetilde{\mathsf{g}}$?, and $\widetilde{\mathsf{t}}$?) performed best overall. In many cases, these even outperformed the provers' native support for simple types ($\widetilde{\mathsf{n}}$). Polymorphic encodings lag behind, especially for resolution provers; this is partly due to the synergy between the monomorphiser and the translation of higher-order constructs (cf. Section 7.2).

- Among the polymorphic encodings, our novel monotonicity-based and cover-based encodings (t?, t??, t@, g?, g??, and g@), with the exception of t@, constitute a substantial improvement over the traditional sound schemes (t and g).
- As suggested in the literature, there is no clear winner between tags and guards. We expected monomorphic guards to be especially effective with SPASS, since they are internally mapped to soft sorts, but this is not corroborated by the data.

- Despite the proof reconstruction phase, the unsound encodings a and $\mathsf{a}^{\mathsf{ctor}}$ achieved similar results to the sound polymorphic encodings. In contrast, their monomorphic cousin $\widetilde{\mathsf{e}}$ is no match for the sound monomorphic schemes.

For the second benchmark set, Figure 4.4 presents the average number of clauses, literals per clause, symbols per atom, and symbols for clausified problems (using E's clausifier), to give an idea of each encoding's overhead. The main surprise here is the lightness of the monomorphic encodings. Because they give rise to duplicate formulas, we could have expected them to result in larger problems; but this underestimates the cost of encoding types as terms in the polymorphic encodings. The strong correlation between the success rates in Figure 4.3 and the average number of symbols in Figure 4.4 confirms the expectation that clutter (whether type arguments, guards, or tags) slows down automatic provers.

Independently of these empirical results, the new type encodings made an impact at the 2012 edition of CASC, the annual automatic prover competition [Sutcliffe, 2012]. Isabelle's automatic proof tools, including Sledgehammer, compete against the automatic provers LEO-II, Satallax, and TPS in the higher-order division. Largely thanks to the new schemes (but also to improvements in the underlying first-order provers), Isabelle moved from the third place it had occupied since 2009 to the first place.

9 Related Work

The earliest descriptions of type tags and type guards we are aware of are due to Enderton [2001, §4.3] and Stickel [1986, p. 99]. Wick and McCune [1989, §4] compare type arguments, tags, and guards in a monomorphic setting. Type arguments are reminiscent of System F; they are described by Meng and Paulson [2008], who also consider full type erasure and polymorphic type tags and present a translation of axiomatic type classes. As part of the MPTP project, Urban [2006] extended the untyped TPTP FOF syntax with dependent types to accommodate Mizar and designed translations to plain FOF.

The intermediate verification language and tool Boogie 2 [Leino and Rümmer, 2010] supports a restricted form of higher-rank polymorphism (with polymorphic maps), and its cousin Why3 [Bobot et al., 2011] provides rank-1 polymorphism. Both define translations to a monomorphic logic and rely on proxies to handle interpreted types [Bobot

and Paskevich, 2011, Couchot and Lescuyer, 2007, Leino and Rümmer, 2010]. One of the Boogie translations [Leino and Rümmer, 2010, §3.1] uses SMT triggers to prevent ill-typed instantiations in conjunction with type arguments; however, this approach is risky in the absence of a semantics for triggers. Bouillaguet et al. [2007, §4] showed that full type erasure is sound if all types can be assumed to have the same cardinality and exploit this in the verification system Jahob.

An alternative to encoding polymorphic types or monomorphising them away is to support them natively in the prover. This is ubiquitous in interactive theorem provers, but perhaps the only automatic prover that supports polymorphism is Alt-Ergo [Bobot et al., 2008].[4]

Blanchette and Krauss [2011] studied monotonicity inferences for higher-order logic without polymorphism. Claessen et al. (Paper 3) were first to apply them to type erasure.

10 Conclusion

This paper introduced a family of translations from polymorphic into untyped first-order logic, with a focus on efficiency. Our monotonicity-based encodings soundly erase all types that are inferred monotonic, as well as most occurrences of the remaining types. The best translations outperform the traditional encoding schemes.

We implemented the new translations in the Sledgehammer tool for Isabelle/HOL and the companion proof method *metis*, thereby addressing a recurring user complaint. Although Isabelle certifies external proofs, unsound proofs are annoying and often conceal sound proofs. The same translation module forms the core of Isabelle's TPTP exporter tool, which makes entire theorem libraries available to first-order reasoners. Our refinements to the monomorphic case have made their way into the Monotonox translatorof Paper 3. Applications such as Boogie [Leino and Rümmer, 2010], LEO-II [Benzmüller et al., 2008], and Why3 [Bobot et al., 2011] also stand to gain from lighter encodings.

The TPTP family recently welcomed the addition of TFF1 [Blanchette and Paskevich, 2012], an extension of the monomorphic TFF0 logic with rank-1 polymorphism. Equipped with a concrete syntax and translation tools, we can turn any popular automatic theorem prover into an efficient polymorphic prover. Translating the untyped proof back into a typed proof is usually straightforward, but there are important

[4] Regrettably, limitations in its type system make it unsuitable as a Sledgehammer backend.

corner cases that call for more research. It should also be possible to extend our approach to interpreted arithmetic.

From both a conceptual and an implementation point of view, the encodings are all instances of a general framework, in which mostly orthogonal features can be combined in various ways. Defining such a large number of encodings makes it possible to select the most appropriate scheme for each automatic prover, based on empirical evidence. In fact, using time slicing or parallelism, it pays off to have each prover employ a combination of encodings with complementary strengths.

Acknowledgement. Koen Claessen and Tobias Nipkow made this collaboration possible. Lukas Bulwahn, Peter Lammich, Rustan Leino, Tobias Nipkow, Mark Summerfield, Tjark Weber, and several anonymous reviewers suggested dozens of textual improvements. We thank them all. The first author's research was supported by the Deutsche Forschungsgemeinschaft (grants Ni 491/11-2 and Ni 491/14-1). The authors are listed in alphabetical order regardless of individual contributions or seniority.

5

Finding Race Conditions in Erlang with QuickCheck and PULSE

This paper was published at ICFP 2009 in Edinburgh.

Chapter 5

Finding Race Conditions in Erlang with QuickCheck and PULSE

Thomas Arts Koen Claessen John Hughes
Michał Pałka Nicholas Smallbone
Hans Svensson Ulf Wiger

Abstract

We address the problem of testing and debugging concurrent, distributed Erlang applications. In concurrent programs, race conditions are a common class of bugs and are very hard to find in practice. Traditional unit testing is normally unable to help finding all race conditions, because their occurrence depends so much on timing. Therefore, race conditions are often found during system testing, where due to the vast amount of code under test, it is often hard to diagnose the error resulting from race conditions. We present three tools (QuickCheck, PULSE, and a visualizer) that in combination can be used to test and debug concurrent programs in unit testing with a much better possibility of detecting race conditions. We evaluate our method on an industrial concurrent case study and illustrate how we find and analyze the race conditions.

1 Introduction

Concurrent programming is notoriously difficult, because the non-deterministic interleaving of events in concurrent processes can lead

software to work most of the time, but fail in rare and hard-to-reproduce circumstances when an unfortunate order of events occurs. Such failures are called *race conditions*. In particular, concurrent software may work perfectly well during *unit testing*, when individual modules (or "software units") are tested in isolation, but fail later on during *system testing*. Even if unit tests cover all aspects of the units, we still can detect concurrency errors when all components of a software system are tested together. Timing delays caused by other components lead to new, previously untested, schedules of actions performed by the individual units. In the worst case, bugs may not appear until the system is put under heavy load in production. Errors discovered in these late stages are far more expensive to diagnose and correct, than errors found during unit testing. Another cause of concurrency errors showing up at a late stage is when well-tested software is ported from a single-core to a multi-core processor. In that case, one would really benefit from a hierarchical approach to testing legacy code in order to simplify debugging of faults encountered.

The Erlang programming language [Armstrong, 2007] is designed to simplify concurrent programming. Erlang processes do not share memory, and Erlang data structures are immutable, so the kind of *data races* which plague imperative programs, in which concurrent processes race to read and write the same memory location, simply cannot occur. However, this does not mean that Erlang programs are immune to race conditions. For example, the order in which messages are delivered to a process may be non-deterministic, and an unexpected order may lead to failure. Likewise, Erlang processes can share *data*, even if they do not share memory—the file store is one good example of shared mutable data, but there are also shared data-structures managed by the Erlang virtual machine, which processes can race to read and write.

Industrial experience is that the late discovery of race conditions is a real problem for Erlang developers too [Cronqvist, 2004]. Moreover, these race conditions are often caused by design errors, which are particularly expensive to repair. If these race conditions could be found during unit testing instead, then this would definitely reduce the cost of software development.

In this paper, we describe tools we have developed for finding race conditions in Erlang code during unit testing. Our approach is based on *property-based testing* using QuickCheck [Claessen and Hughes, 2000], in a commercial version for Erlang developed by Quviq AB [Arts et al., 2006, Hughes, 2007]. Its salient features are described in section 3. We develop a suitable property for testing parallel code, and a method

for generating parallel test cases, in section 4. To test a wide variety of schedules, we developed a randomizing scheduler for Erlang called PULSE, which we explain in section 5. PULSE records a trace during each test, but interpreting the traces is difficult, so we developed a trace visualizer which is described in section 6. We evaluate our tools by applying them to an industrial case study, which is introduced in section 2, then used as a running example throughout the paper. This code was already known to contain bugs (thanks to earlier experiments with QuickCheck in 2005), but we were previously unable to *diagnose* the problems. Using the tools described here, we were able to find and fix two race conditions, and identify a fundamental flaw in the API.

2 Our Case Study: the Process Registry

We begin by introducing the industrial case that we apply our tools and techniques to. In Erlang, each process has a unique, dynamically-assigned identifier ("pid"), and to send a message to a process, one must know its pid. To enable processes to discover the pids of central services, such as error logging, Erlang provides a *process registry*—a kind of local name server—which associates static names with pids. The Erlang VM provides operations to *register* a pid with a name, to *look up* the pid associated with a name, and to *unregister* a name, removing any association with that name from the registry. The registry holds only *live* processes; when registered processes crash, then they are automatically unregistered. The registry is heavily used to provide access to system services: a newly started Erlang node already contains 13 registered processes.

However, the built-in process registry imposes several, sometimes unwelcome, limitations: registered names are restricted to be atoms, the same process cannot be registered with multiple names, and there is no efficient way to search the registry (other than by name lookup). This motivated Ulf Wiger (who was working for Ericsson at the time) to develop an extended process registry *in Erlang*, which could be modified and extended much more easily than the one in the virtual machine. Wiger's process registry software has been in use in Ericsson products for several years [Wiger, 2007].

In our case study we consider an earlier prototype of this software, called `proc_reg`, incorporating an optimization that proved not to work. The API supported is just: `reg(Name,Pid)` to register a pid, `where(Name)` to look up a pid, `unreg(Name)` to remove a registration, and finally

`send(Name,Msg)` to send a message to a registered process. Like the production code, `proc_reg` stores the association between names and pids in *Erlang Term Storage* ("ETS tables")—hash tables, managed by the virtual machine, that hold a set of tuples and support tuple-lookup using the first component as a key [cf. Armstrong, 2007, chap 15]. It also creates a *monitor* for each registered process, whose effect is to send `proc_reg` a "`DOWN`" message if the registered process crashes, so it can be removed from the registry. Two ETS table entries are created for each registration: a "forward" entry that maps names to pids, and a "reverse" entry that maps registered pids to the monitor reference. The monitor reference is needed to turn off monitoring again, if the process should later be unregistered.

Also like the production code, `proc_reg` is implemented as a server process using Erlang's *generic server* library [cf. Armstrong, 2007, chap 16]. This library provides a robust way to build client-server systems, in which clients make "synchronous calls" to the server by sending a `call` message, and awaiting a matching reply[1]. Each operation—`reg`, `where`, `unreg` and `send`—is supported by a different `call` message. The operations are actually executed by the server, one at a time, and so no race conditions can arise.

At least, this is the theory. In practice there is a small cost to the generic server approach: each request sends two messages and requires two context switches, and although these are cheap in Erlang, they are not free, and turn out to be a bottleneck in system start-up times, for example. The prototype `proc_reg` attempts to optimize this, by moving the creation of the first "forward" ETS table entry into the clients. If this succeeds (because there is no previous entry with that name), then clients just make an "asynchronous" call to the server (a so-called `cast` message, with no reply) to inform it that it should complete the registration later. This avoids a context switch, and reduces two messages to one. If there *is* already a registered process with the same name, then the `reg` operation fails (with an exception)—unless, of course, the process is dead. In this case, the process will soon be removed from the registry by the server; clients ask the server to "audit" the dead process to hurry this along, then complete their registration as before.

This prototype was one of the first pieces of software to be tested using QuickCheck at Ericsson. At the time, in late 2005, it was believed to work, and indeed was accompanied by quite an extensive suite of unit

[1] Unique identifiers are generated for each call, and returned in the reply, so that no message confusion can occur.

tests—including cases designed specifically to test for race conditions. We used QuickCheck to generate and run random sequences of API calls in two concurrent processes, and instrumented the `proc_reg` code with calls to `yield()` (which cedes control to the scheduler) to cause fine-grain interleaving of concurrent operations. By so doing, we could show that `proc_reg` was incorrect, since our tests failed. But the failing test cases we found were large, complex, and very hard to understand, and we were unable to use them to diagnose the problem. As a result, this version of `proc_reg` was abandoned, and development of the production version continued without the optimization.

While we were pleased that QuickCheck revealed bugs in `proc_reg`, we were unsatisfied that it could not help us to find them. Moreover, the QuickCheck property we used to test it was hard-to-define and ad hoc—and not easily reusable to test any other software. This paper is the story of how we addressed these problems—and returned to apply our new methods successfully to the example that defeated us before.

3 An Overview of Quviq QuickCheck

QuickCheck [Claessen and Hughes, 2000] is a tool that tests universally quantified *properties*, instead of single test cases. QuickCheck generates random test cases from each property, tests whether the property is true in that case, and reports cases for which the property fails. Recent versions also "shrink" failing test cases automatically, by searching for similar, but smaller test cases that also fail. The result of shrinking is a "minimal"[2] failing case, which often makes the root cause of the problem very easy to find.

Quviq QuickCheck is a commercial version that includes support for model-based testing using a state machine model [Hughes, 2007]. This means that it has standard support for generating sequences of API calls using this state machine model. It has been used to test a wide variety of industrial software, such as Ericsson's Media Proxy [Arts et al., 2006] among others. State machine models are tested using an additional library, `eqc_statem`, which invokes call-backs supplied by the user to generate and test random, well-formed sequences of calls to the software under test. We illustrate `eqc_statem` by giving fragments of a (sequential) specification of `proc_reg`.

[2] In the sense that it cannot shrink to a failing test with the shrinking algorithm used.

Let us begin with an example of a generated test case (a sequence of API calls).

```
[{set,{var,1},{call,proc_reg_eqc,spawn,[]}},
 {set,{var,2},{call,proc_reg,where,[c]}},
 {set,{var,3},{call,proc_reg_eqc,spawn,[]}},
 {set,{var,4},{call,proc_reg_eqc,kill,[{var,1}]}},
 {set,{var,5},{call,proc_reg,where,[d]}},
 {set,{var,6},{call,proc_reg_eqc,reg,[a,{var,1}]}},
 {set,{var,7},{call,proc_reg_eqc,spawn,[]}}]
```

`eqc_statem` test cases are lists of *symbolic commands* represented by Erlang terms, each of which binds a symbolic variable (such as `{var,1}`) to the result of a function call, where `{call,M,F,Args}` represents a call of function `F` in module `M` with arguments `Args`[3]. Note that previously bound variables can be used in later calls. Test cases for `proc_reg` in particular randomly spawn processes (to use as test data), kill them (to simulate crashes at random times), or pass them to `proc_reg` operations. Here `proc_reg_eqc` is the module containing the specification of `proc_reg`, in which we define local versions of `reg` and `unreg` which just call `proc_reg` and catch any exceptions. This allows us to write properties that test whether an exception is raised correctly or not. (An *uncaught* exception in a test is interpreted as a failure of the entire test).

We model the state of a test case as a list of processes spawned, processes killed, and the `{Name,Pid}` pairs currently in the registry. We normally encapsulate the state in a record:

```
-record(state,{pids=[],regs=[],killed=[]}).
```

`eqc_statem` generates random calls using the call-back function `command` that we supply as part of the state machine model, with the test case state as its argument:

```
command(S) ->
  oneof(
    [{call,?MODULE,spawn,[]}] ++
    [{call,?MODULE,kill, [elements(S#state.pids)]}
           || S#state.pids/=[]] ++
    [{call,?MODULE,reg,[name(),elements(S#state.pids)]}
           || S#state.pids/=[]] ++
    [{call,?MODULE,unreg,[name()]}] ++
    [{call,proc_reg,where,[name()]}]).

name() -> elements([a,b,c,d]).
```

[3] In Erlang, variables start with an uppercase character, whereas atoms (constants) start with a lowercase character.

The function oneof is a QuickCheck generator that randomly uses one element from a list of generators; in this case, the list of candidates to choose from depends on the test case state. ([X||P] is a degenerate list comprehension, that evaluates to the empty list if P is false, and [X] if P is true—so reg and kill can be generated only if there are pids available to pass to them.) We decided not to include send in test cases, because its implementation is quite trivial. The macro ?MODULE expands to the name of the module that it appears in, proc_reg_eqc in this case.

The next_state function specifies how each call is supposed to change the state:

```
next_state(S,V,{call,_,spawn,_}) ->
  S#state{pids=[V|S#state.pids]};
next_state(S,V,{call,_,kill,[Pid]}) ->
  S#state{killed=[Pid|S#state.killed],
          regs=[{Name,P} ||
                {Name,P} <- S#state.regs, Pid /= P]};
next_state(S,_V,{call,_,reg,[Name,Pid]}) ->
  case register_ok(S,Name,Pid) andalso
         not lists:member(Pid,S#state.killed) of
       true ->
         S#state{regs=[{Name,Pid}|S#state.regs]};
       false ->
         S
  end;
next_state(S,_V,{call,_,unreg,[Name]}) ->
  S#state{regs=lists:keydelete(Name,1,S#state.regs)};
next_state(S,_V,{call,_,where,[_]}) ->
  S.

register_ok(S,Name,Pid) ->
  not lists:keymember(Name,1,S#state.regs).
```

Note that the new state can depend on the *result* of the call (the second argument V), as in the first clause above. Note also that killing a process removes it from the registry (in the model), and that registering a dead process, or a name that is already registered (see register_ok), should not change the registry state. We do allow the same pid to be registered with several names, however.

When running tests, eqc_statem checks the postcondition of each call, specified via another call-back that is given the state before the call, and the actual result returned, as arguments. Since we catch exceptions in each call, which converts them into values of the form {'EXIT',Reason}, our proc_reg postconditions can test that exceptions are raised under precisely the right circumstances:

```
postcondition(S,{call,_,reg,[Name,Pid]},Res) ->
  case Res of
       true ->
         register_ok(S,Name,Pid);
       {'EXIT',_} ->
         not register_ok(S,Name,Pid)
  end;
postcondition(S,{call,_,unreg,[Name]},Res) ->
  case Res of
       true ->
         unregister_ok(S,Name);
       {'EXIT',_} ->
         not unregister_ok(S,Name)
  end;
postcondition(S,{call,_,where,[Name]},Res) ->
  lists:member({Name,Res},S#state.regs);
postcondition(_S,{call,_,_,_},_Res) ->
  true.

unregister_ok(S,Name) ->
  lists:keymember(Name,1,S#state.regs).
```

Note that `reg(Name,Pid)` and `unreg(Name)` are required to return exceptions if `Name` is already used/not used respectively, but that `reg` always returns `true` if `Pid` is dead, even though no registration is performed! This may perhaps seem a surprising design decision, but it is consistent. As a comparison, the built-in process registry sometimes returns `true` and sometimes raises an exception when registering dead processes. This is due to the fact that a context switch is required to clean up.

State machine models can also specify a *precondition* for each call, which restricts test cases to those in which all preconditions hold. In this example, we could have used preconditions to exclude test cases that we expect to raise exceptions—but we prefer to allow any test case, and check that exceptions are raised correctly, so we define all preconditions to be `true`.

With these four call-backs, plus another call-back specifying the initial state, our specification is almost complete. It only remains to define the top-level property which generates and runs tests:

```
prop_proc_reg() ->
  ?FORALL(Cmds,commands(?MODULE),
     begin
       {ok,ETSTabs} = proc_reg_tabs:start_link(),
       {ok,Server} = proc_reg:start_link(),
       {H,S,Res} = run_commands(?MODULE,Cmds),
```

```
    cleanup(ETSTabs,Server),
    Res == ok
  end).
```

Here `?FORALL` binds `Cmds` to a random list of commands generated by `commands`, then we initialize the registry, run the commands, clean up, and check that the result of the run (`Res`) was a success. Here `commands` and `run_commands` are provided by `eqc_statem`, and take the current module name as an argument in order to find the right call-backs. The other components of `run_commands`' result, `H` and `S`, record information about the test run, and are of interest primarily when a test fails. This is not the case here: *sequential* testing of `proc_reg` does not fail.

4 Parallel Testing with QuickCheck

4.1 A Parallel Correctness Criterion

In order to test for race conditions, we need to generate test cases that are executed in parallel, and we also need *a specification of the correct parallel behavior*. We have chosen, in this paper, to use a specification that just says that *the API operations we are testing should behave atomically*.

How can we tell from test results whether or not each operation "behaved atomically"? Following Lamport [1979] and Herlihy and Wing [1987], we consider a test to have passed if the observed results are the same as some possible sequential execution of the operations in the test—that is, a possible interleaving of the parallel processes in the test.

Of course, testing for atomic behavior is just a special case, and in general we may need to test other properties of concurrent code too—but we believe that this is a very important special case. Indeed, Herlihy and Wing claim that their notion of *linearizability* "focuses exclusively on a subset of concurrent computations that we believe to be the most interesting and useful"; we agree. In particular, atomicity is of great interest for the process registry.

One great advantage of this approach is that we can reuse the *same* specification of the sequential behavior of an API, to test its behavior when invocations take place in parallel. We need only find the right linearization of the API calls in the test, and then use the sequential specification to determine whether or not the test has passed. We have implemented this idea in a new QuickCheck module, `eqc_par_statem`,

which takes the *same* state-machine specifications as `eqc_statem`, but tests the API in parallel instead. While state machine specifications require some investment to produce in real situations, this means that we can test for race conditions *with no further investment* in developing a parallel specification. It also means that, as the code under test evolves, we can switch freely to-and-fro between sequential testing to ensure the basic behavior still works, and race condition testing using `eqc_par_statem`.

The difficulty with this approach is that, when we run a test, then there is no way to *observe* the sequence in which the API operations take effect. (For example, a server is under no obligation to service requests in the order in which they are made, so observing this order would tell us nothing.) In general, the only way to tell whether there is a possible sequentialization of a test case which can explain the observed test results, is to *enumerate* all possible sequentializations. This is prohibitively expensive unless care is taken when test cases are generated.

4.2 Generating Parallel Test Cases

Our first approach to parallel test case generation was to use the standard Quviq QuickCheck library `eqc_statem` to generate sequential test cases, then execute all the calls in the test case in parallel, constrained only by the data dependencies between them (which arise from symbolic variables, bound in one command, being used in a later one). This generates a great deal of parallelism, but sadly also an enormous number of possible serializations—in the worst case in which there are no data dependencies, a sequence of n commands generates $n!$ possible serializations. It is not practically feasible to implement a test oracle for parallel tests of this sort.

Instead, we decided to generate parallel test cases of a more restricted form. They consist of an initial sequential prefix, to put the system under test into a random state, followed by exactly *two* sequences of calls which are performed in parallel. Thus the possible serializations consist of the initial prefix, followed by an interleaving of the two parallel sequences. (Lu et al. [2008] gives clear evidence that it is possible to discover a large fraction of the concurrency related bugs by using only two parallel threads/processes.) We generate parallel test cases by parallelizing a suffix of an `eqc_statem` test case, separating it into two lists of commands of roughly equal length, with no mutual data dependencies, which are *non-interfering* according to the sequential specification. By

non-interference, we mean that all command preconditions are satisfied in any interleaving of the two lists, which is necessary to prevent tests from failing because a precondition was unsatisfied—not an interesting failure. We avoid parallelizing too long a suffix (longer than 16 commands), to keep the number of possible interleavings feasible to enumerate (about 10,000 in the worst case). Finally, we run tests by first running the prefix, then spawning two processes to run the two command-lists in parallel, and collecting their results, which will be non-deterministic depending on the actual parallel scheduling of events.

We decide whether a test has passed, by attempting to construct a sequentialization of the test case which explains the results observed. We begin with the sequential prefix of the test case, and use the `next_state` function of the `eqc_statem` model to compute the test case state after this prefix is completed. Then we try to *extend* the sequential prefix, one command at a time, by choosing the first command from one of the parallel branches, and moving it into the prefix. This is allowed only if the `postcondition` specified in the `eqc_statem` model accepts the actual result returned by the command when we ran the test. If so, we use the `next_state` function to compute the state after this command, and continue. If the first commands of *both* branches fulfilled their postconditions, then we cannot yet determine which command took effect first, and we must explore both possibilities further. If we succeed in moving *all* commands from the parallel branches into the sequential prefix, such that all postconditions are satisfied, then we have found a possible sequentialization of the test case explaining the results we observed. If our search fails, then there is *no* such sequence, and the test failed.

This is a greedy algorithm: as soon as a postcondition fails, then we can discard all potential sequentializations with the failing command as the next one in the sequence. This happens often enough to make the search reasonably fast in practice. As a further optimization, we memoize the search function on the remaining parallel branches and the test case state. This is useful, for example, when searching for a sequentialization of [A, B] and [C, D], if both [A, C] and [C, A] are possible prefixes, and they lead to the same test state—for then we need only try to sequentialize [B] and [D] once. We memoize the non-interference test in a similar way, and these optimizations give an appreciable, though not dramatic, speed-up in our experiments—of about 20%. With these optimizations, generating and running parallel tests is acceptably fast.

4.3 Shrinking Parallel Test Cases

When a test fails, QuickCheck attempts to *shrink* the failing test by searching for a similar, but smaller test case which also fails. QuickCheck can often report minimal failing examples, which is a great help in fault diagnosis. `eqc_statem` already has built-in shrinking methods, of which the most important tries to delete unnecessary commands from the test case, and `eqc_par_statem` inherits these methods. But we also implement an additional shrinking method for parallel test cases: if it is possible to move a command from one of the parallel suffixes into the sequential prefix, then we do so. Thus the minimal test cases we find are "minimally parallel"—we know that the parallel branches in the failing tests reported really do race, because everything that can be made sequential, is sequential. This also assists fault diagnosis.

4.4 Testing `proc_reg` for Race Conditions

To test the process registry using `eqc_par_statem`, it is only necessary to modify the property in Section 2 to use `eqc_par_statem` rather than `eqc_statem` to generate and run test cases.

```
prop_proc_reg_parallel() ->
  ?FORALL(Cmds,eqc_par_statem:commands(?MODULE),
     begin
       {ok,ETSTabs} = proc_reg_tabs:start_link(),
       {ok,Server} = proc_reg:start_link(),
       {H,{A,B},Res} =
         eqc_par_statem:run_commands(?MODULE,Cmds),
       cleanup(ETSTabs,Server),
       Res == ok
     end).
```

The type returned by `run_commands` is slightly different (`A` and `B` are lists of the calls made in each parallel branch, paired with the results returned), but otherwise no change to the property is needed.

When this property is tested on a single-core processor, all tests pass. However, as soon as it is tested on a dual-core, tests begin to fail. Interestingly, just running on two cores gives us enough fine-grain interleaving of concurrent processes to demonstrate the presence of race conditions, something we had to achieve by instrumenting the code with calls to `yield()` to control the scheduler when we first tested this code in 2005. However, just as in 2005, the reported failing test cases are large, and do not shrink to small examples. This makes the race condition very hard indeed to diagnose.

The problem is that the test outcome is not determined solely by the test case: depending on the actual interleaving of memory operations on the dual core, the same test may sometimes pass and sometimes fail. This is devastating for QuickCheck's shrinking, which works by repeatedly replacing the failed test case by a smaller one which still fails. If the smaller test happens to succeed—by sheer chance, as a result of non-deterministic execution—then the shrinking process stops. This leads QuickCheck to report failed tests which are far from minimal.

Our solution to this is almost embarrassing in its simplicity: instead of running each test only once, we run it many times, and consider a test case to have passed only if it passes every time we run it. We express this concisely using a new form of QuickCheck property, `?ALWAYS(N,Prop)`, which passes if `Prop` passes `N` times in a row[4]. Now, provided the race condition we are looking for is reasonably likely to be provoked by test cases in which it is present, then `?ALWAYS(10,...)` is very likely to provoke it—and so tests are unlikely to succeed "by chance" during the shrinking process. This dramatically improves the effectiveness of shrinking, even for quite small values of `N`. While we do not *always* obtain minimal failing tests with this approach, we find we can usually obtain a minimal example by running QuickCheck a few times.

When testing the `proc_reg` property above, we find the following simple counterexample:

```
{[{set,{var,5},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,9},{call,proc_reg_eqc,kill,[{var,5}]}},
  {set,{var,15},{call,proc_reg_eqc,reg,[a,{var,5}]}}],
 {[{set,{var,19},{call,proc_reg_eqc,reg,[a,{var,5}]}}],
  [{set,{var,18},{call,proc_reg_eqc,reg,[a,{var,5}]}}]}}
```

This test case first creates and kills a process, then tries to register it (which should have no effect, because it is already dead), and finally tries to register it again twice, in parallel. Printing the diagnostic output from `run_commands`, we see:

```
Sequential:
  [{{state,[],[],[]},<0.5576.2>},
   {{state,[<0.5576.2>],[],[]},ok},
   {{state,[<0.5576.2>],[],[<0.5576.2>]},true}]
Parallel:
  {[{{call,proc_reg_eqc,reg,[a,<0.5576.2>]},
     {'EXIT',{badarg,[{proc_reg,reg,2},...]}}}],
   [{{call,proc_reg_eqc,reg,[a,<0.5576.2>]},true}]}
Res: no_possible_interleaving
```

[4] In fact we need only repeat tests during shrinking.

(where the ellipses replace an uninteresting stack trace). The values displayed under "`Parallel:`" are the results `A` and `B` from the two parallel branches—they reveal that one of the parallel calls to `reg` raised an exception, even though trying to register a dead process should always just return `true`! How this happened, though, is still quite mysterious—but will be explained in the following sections.

5 PULSE: A User-level Scheduler

At this point, we have found a simple test case that fails, but we do not know *why* it failed—we need to debug it. A natural next step would be to turn on Erlang's tracing features and rerun the test. But when the bug is caused by a race condition, then turning on tracing is likely to change the timing properties of the code, and thus interfere with the test failure! Even simply repeating the test may lead to a different result, because of the non-determinism inherent in running on a multi-core. This is devastating for debugging.

What we need is to be able to *repeat* the test as many times as we like, with deterministic results, and to *observe* what happens during the test, so that we can analyze how the race condition was provoked. With this in mind, we have implemented a new Erlang module that can control the execution of designated Erlang processes and records a trace of all relevant events. Our module can be thought of as a *user-level scheduler*, sitting on top of the normal Erlang scheduler. Its aim is to take control over all sources of non-determinism in Erlang programs, and instead take those scheduling decisions randomly. This means that we can repeat a test using exactly the same schedule by supplying the same random number seed: this makes tests repeatable. We have named the module PULSE, short for *ProTest* User-Level Scheduler for Erlang.

The Erlang virtual machine (VM) runs processes for relatively long time-slices, in order to minimize the time spent on context switching—but as a result, it is very unlikely to provoke race conditions in small tests. It is possible to tune the VM to perform more context switches, but even then the scheduling decisions are entierly deterministic. This is one reason why tricky concurrency bugs are rarely found during unit testing; it is not until later stages of a project, when many components are tested together, that the standard scheduler begins to preempt processes and trigger race conditions. In the worst case, bugs don't appear until the system is put under heavy load in production! In these later stages, such errors are expensive to debug. One other advantage (apart

from repeatability) of PULSE is that it generates much more fine-grain interleaving than the built-in scheduler in the Erlang virtual machine (VM), because it randomly chooses the next process to run at each point. Therefore, we can provoke race conditions even in very small tests.

Erlang's scheduler is built into its virtual machine—and we did *not* want to modify the virtual machine itself. Not only would this be difficult—it is a low-level, fairly large and complex C program—but we would need to repeat the modifications every time a new version of the virtual machine was released. We decided, therefore, to implement PULSE in Erlang, as a user-level scheduler, and to *instrument* the code of the processes that it controls so that they cooperate with it. As a consequence, PULSE can even be used in conjunction with legacy or customized versions of the Erlang VM (which are used in some products). The user level scheduler also allows us to restrict our debugging effort to a few processes, whereas we are guaranteed that the rest of the processes are executed normally.

5.1 Overall Design

The central idea behind developing PULSE was to provide absolute control over the order of relevant events. The first natural question that arises is: What are the relevant events? We define a *side-effect* to be any interaction of a process with its environment. Of particular interest in Erlang is the way processes interact by message passing, which is asynchronous. Message channels, containing messages that have been sent but not yet delivered, are thus part of the environment and explicitly modelled as such in PULSE. It makes sense to separate side-effects into two kinds: *outward* side-effects, that influence only the environment (such as sending a message over a channel, which does not block and cannot fail, or printing a message), and *inward* side-effects, that allow the environment to influence the behavior of the process (such as receiving a message from a channel, or asking for the system time).

We do not want to take control over purely functional code, or side-effecting code that only influences processes locally. PULSE takes control over some basic features of the Erlang RTS (such as spawning processes, message sending, linking, etc.), but it knows very little about standard library functions – it would be too much work to deal with each of these separately! Therefore, the user of PULSE can specify which library functions should be dealt with as (inward) side-effecting

functions, and PULSE has a generic way of dealing with these (see subsection 5.3).

A process is only under the control of PULSE if its code has been properly instrumented. All other processes run as normal. In instrumentation, occurrences of side-effecting actions are replaced by indirections that communicate with PULSE instead. In particular, outward side-effects (such as sending a message to another process) are replaced by simply sending a message to PULSE with the details of the side-effect, and inward side-effects (such as receiving a message) are replaced by sending a request to PULSE for performing that side-effect, and subsequently waiting for permission. To ease the instrumentation process, we provide an automatic instrumenter, described in subsection 5.4.

5.2 Inner Workings

The PULSE scheduler controls its processes by allowing only one of them to run at a time. It employs a cooperative scheduling method: At each decision point, PULSE randomly picks one of its waiting processes to proceed, and wakes it up. The process may now perform a number of outward side-effects, which are all recorded and taken care of by PULSE, until the process wants to perform an inward side-effect. At this point, the process is put back into the set of waiting processes, and a new decision point is reached.

The (multi-node) Erlang semantics [Svensson and Fredlund, 2007] provides only one guarantee for message delivery order: that messages between a **pair** of processes arrive in the same order as they were sent. So as to adhere to this, PULSE's state also maintains a message queue between each pair of processes. When process P performs an outward side-effect by sending a message M to the process Q, then M is added to the queue $\langle P, Q \rangle$. When PULSE wants to wake up a waiting process Q, it does so by randomly picking a non-empty queue $\langle P', Q \rangle$ with Q as its destination, and delivering the first message in that queue to Q. Special care needs to be taken for the Erlang construct `receive ... after n -> ... end`, which allows a receiving process to only wait for an incoming message for n milliseconds before continuing, but the details of this are beyond the scope of this paper.

As an additional benefit, this design allows PULSE to detect deadlocks when it sees that all processes are blocked, and there exist no message queues with the blocked processes as destination.

As a clarification, the message queues maintained by PULSE for each pair of processes should not be confused with the internal mailbox

that each process in Erlang has. In our model, sending a message M from P to Q goes in four steps: (1) P asynchronously sends off M, (2) M is on its way to Q, (3) M is delivered to Q's mailbox, (4) Q performs a `receive` statement and M is selected and removed from the mailbox. The only two events in this process that we consider side-effects are (1) P sending of M, and (3) delivering M to Q's mailbox. In what order a process decides to process the messages in its mailbox is not considered a side-effect, because no interaction with the environment takes place.

5.3 External Side-effects

In addition to sending and receiving messages between themselves, the processes under test can also interact with uninstrumented code. PULSE needs to be able to control the order in which those interactions take place. Since we are not interested in controlling the order in which pure functions are called we allow the programmer to specify which external functions have side-effects. Each call of a side-effecting function is then instrumented with code that `yield`s before performing the real call and PULSE is free to run another process at that point.

Side-effecting functions are treated as atomic which is also an important feature that aids in testing systems built of multiple components. Once we establish that a component contains no race conditions we can remove the instrumentation from it and mark its operations as atomic side-effects. We will then be able to test other components that use it and each operation marked as side-effecting will show up as a single event in a trace. Therefore, it is possible to test a component for race conditions independently of the components that it relies on.

5.4 Instrumentation

The program under test has to cooperate with PULSE, and the relevant processes should use PULSE's API to send and receive messages, spawn processes, etc., instead of Erlang's built-in functionality. Manually altering an Erlang program so that it does this is tedious and error-prone, so we developed an instrumenting compiler that does this automatically. The instrumenter is used in exactly the same way as the normal compiler, which makes it easy to switch between PULSE and the normal Erlang scheduler. It's possible to instrument and load a module at runtime by typing in a single command at the Erlang shell.

Let us show the instrumentation of the four most important constructs: sending a message, yielding, spawning a process, and receiving

a message.

Sending

If a process wants to send a message, the instrumenter will redirect this as a request to the PULSE scheduler. Thus, `Pid ! Msg` is replaced by

```
scheduler ! {send, Pid, Msg},
Msg
```

The result value of sending a message is always the message that was sent. Since we want the instrumented *send* to yield the same result value as the original one, we add the second line.

Yielding

A process yields when it wants to give up control to the scheduler. Yields are also introduced just before each user-specified side-effecting external function.

After instrumentation, a yielding process will instead give up control to PULSE. This is done by telling it that the process yields, and waiting for permission to continue. Thus, `yield()` is replaced by

```
scheduler ! yield,
receive
  {scheduler, go} -> ok
end
```

In other words, the process notifies PULSE and then waits for the message go from the scheduler before it continues. All control messages sent by PULSE to the controlled processes are tagged with `{scheduler, _}` in order to avoid mixing them up with "real" messages.

Spawning

A process P spawning a process Q is considered an outward side-effect for P, and thus P does not have to block. However, PULSE must be informed of the existence of the new process Q, and Q needs to be brought under its control. The spawned process Q must therefore wait for PULSE to allow it to run. Thus, `spawn(Fun)` is replaced by

```
Pid = spawn(fun() -> receive
                       {scheduler, go} -> Fun()
                     end
            end),
scheduler ! {spawned, Pid},
Pid
```

In other words, the process spawns an altered process that waits for the message go from the scheduler before it does anything. The scheduler is then informed of the existence of the spawned process, and we continue.

Receiving

Receiving in Erlang works by pattern matching on the messages in the process' mailbox. When a process is ready to receive a new message, it will have to ask PULSE for permission. However, it is possible that an appropriate message already exists in its mailbox, and receiving this message would not be a side-effect. Therefore, an instrumented process will first check if it is possible to receive a message with the desired pattern, and proceed if this is possible. If not, it will tell the scheduler that it expects a new message in its mailbox, and blocks. When woken up again on the delivery of a new message, this whole process is repeated if necessary.

We need a helper function that implements this checking-waiting loop. It is called `receiving`:

```
receiving(Receiver) ->
  Receiver(fun() ->
    scheduler ! block,
    receive
      {scheduler, go} -> receiving(Receiver)
    end
  end).
```

`receiving` gets a receiver function as an argument. A receiver function is a function that checks if a certain message is in its mailbox, and if not, executes its argument function. The function `receiving` turns this into a loop that only terminates once PULSE has delivered the right message. When the receiver function fails, PULSE is notified by the `block` message, and the process waits for permission to try again.

Code of the form

```
receive Pat -> Exp end
```

is then replaced by

```
receiving(fun (Failed) ->
            receive
              Pat     -> Exp
              after 0 -> Failed()
            end
          end)
```

In the above, we use the standard Erlang idiom (`receive ... after 0 -> ... end`) for checking if a message of a certain type exists. It is easy to see how receive statements with more than one pattern can be adapted to work with the above scheme.

5.5 Testing `proc_reg` with PULSE

To test the `proc_reg` module using both QuickCheck and PULSE, we need to make a few modifications to the QuickCheck property in Section 4.4.

```
prop_proc_reg_scheduled() ->
  ?FORALL(Cmds,eqc_par_statem:commands(?MODULE),
   ?ALWAYS(10,?FORALL(Seed,seed(),
     begin

       SRes =
         scheduler:start([{seed,Seed}],
           fun() ->
             {ok,ETSTabs} = proc_reg_tabs:start_link(),
             {ok,Server} = proc_reg:start_link(),
             eqc_par_statem:run_commands(?MODULE,Cmds),
             cleanup(ETSTabs,Server),
           end),
       {H,AB,Res} = scheduler:get_result(SRes),
       Res == ok
     end))).
```

PULSE uses a random seed, generated by `seed()`. It also takes a function as an argument, so we create a lambda-function which initializes and runs the tests. The result of running the scheduler is a list of things, thus we need to call `scheduler:get_result` to retrieve the actual result from `run_commands`. We should also remember to *instrument* rather than compile all the involved modules. Note that we still use `?ALWAYS` in order to run the same test data with different random seeds, which helps the shrinking process in finding smaller failing test cases that would otherwise be less likely to fail.

When testing this modified property, we find the following counterexample, which is in fact simpler than the one we found in Section 4.4:

```
{[{set,{var,9},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,10},{call,proc_reg_eqc,kill,[{var,9}]}}],
 {[{set,{var,15},{call,proc_reg_eqc,reg,[c,{var,9}]}}],
  [{set,{var,12},{call,proc_reg_eqc,reg,[c,{var,9}]}}]}}
```

When prompted, PULSE provides quite a lot of information about the test case run and the scheduling decisions taken. Below we show an example of such information. However, it is still not easy to explain the counterexample; in the next section we present a method that makes it easier to understand the scheduler output.

```
 -> <'start_link.Pid1'> calls
    scheduler:process_flag [priority,high]
    returning normal.
 -> <'start_link.Pid1'> sends
    '{call,{attach,<0.31626.0>},
           <0.31626.0>,#Ref<0.0.0.13087>}'
    to <'start_link.Pid'>.
 -> <'start_link.Pid1'> blocks.
*** unblocking <'start_link.Pid'>
    by delivering '{call,{attach,<0.31626.0>},
                         <0.31626.0>,
                         #Ref<0.0.0.13087>}'
    sent by <'start_link.Pid1'>.
...
```

6 Visualizing Traces

PULSE records a complete trace of the interesting events during test execution, but these traces are long, and tedious to understand. To help us interpret them, we have, utilizing the popular GraphViz package [Gansner and North, 1999], built a *trace visualizer* that draws the trace as a graph. For example, Figure 5.1 shows the graph drawn for one possible trace of the following program:

```
procA() ->
  PidB = spawn(fun procB/0),
  PidB ! a,
  process_flag(trap_exit, true),
  link(PidB),
  receive
    {'EXIT',_,Why} -> Why
  end.

procB() ->
  receive
    a -> exit(kill)
  end.
```

The function `procA` starts by spawning a process, and subsequently sends it a message `a`. Later, `procA` *links* to the process it spawned,

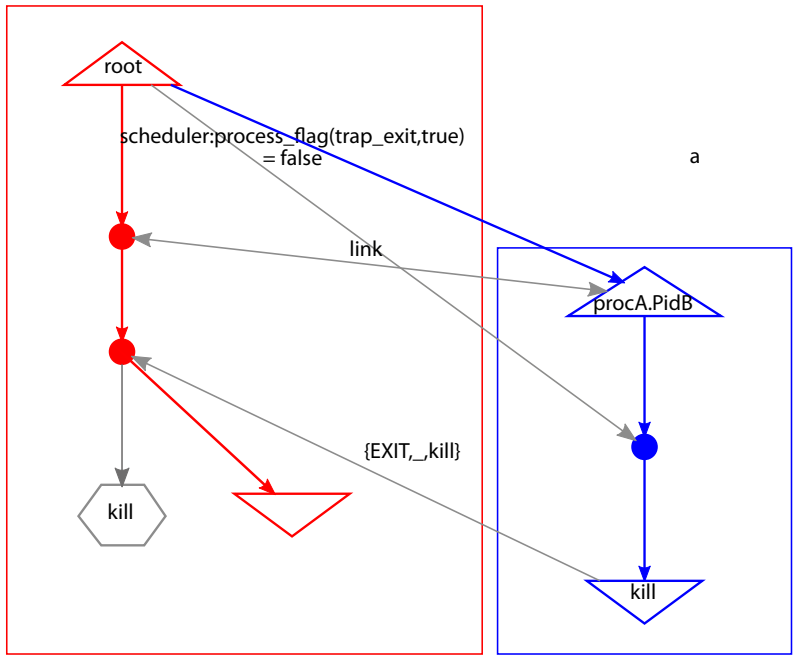


Figure 5.1: A simple trace visualization.

which means that it will get notified when that process dies. The default behavior of a process when such a notification happens is to also die (in this way, one can build hierarchies of processes). Setting the process flag *trap_exit* to true changes this behaviour, and the notification is delivered as a regular message of the form `{EXIT,_,_}` instead.

In the figure, each process is drawn as a sequence of state transitions, from a start state drawn as a triangle, to a final state drawn as an inverted triangle, all enclosed in a box and assigned a unique color. (Since the printed version of the diagrams may lack these colors, we reference diagram details by location and not by color. However, the diagrams are even more clear in color.) The diagram shows the two processes, `procA` (called `root`) which is shown to the left (in red), and `procB` (called `procA.PidB`, a name automatically derived by PULSE from the point at which it was spawned) shown to the right (in blue). Message delivery is shown by gray arrows, as is the return of a result by the `root` process. As explained in the previous section, processes

make transitions when receiving a message[5], or when calling a function that the instrumenter knows has a side-effect. From the figure, we can see that the `root` process spawned `PidB` and sent the message `a` to it, but before the message was delivered then the `root` process managed to set its `trap_exit` process flag, and linked to `PidB`. `PidB` then received its message, and killed itself, terminating with reason `kill`. A message was sent back to `root`, which then returned the exit reason as its result.

Figure 5.2 shows an alternative trace, in which `PidB` dies *before* `root` creates a link to it, which generates an exit message with a different exit reason. The existence of these two different traces indicates a race condition when using `spawn` and `link` separately (which is the reason for the existence of an atomic `spawn_link` function in Erlang).

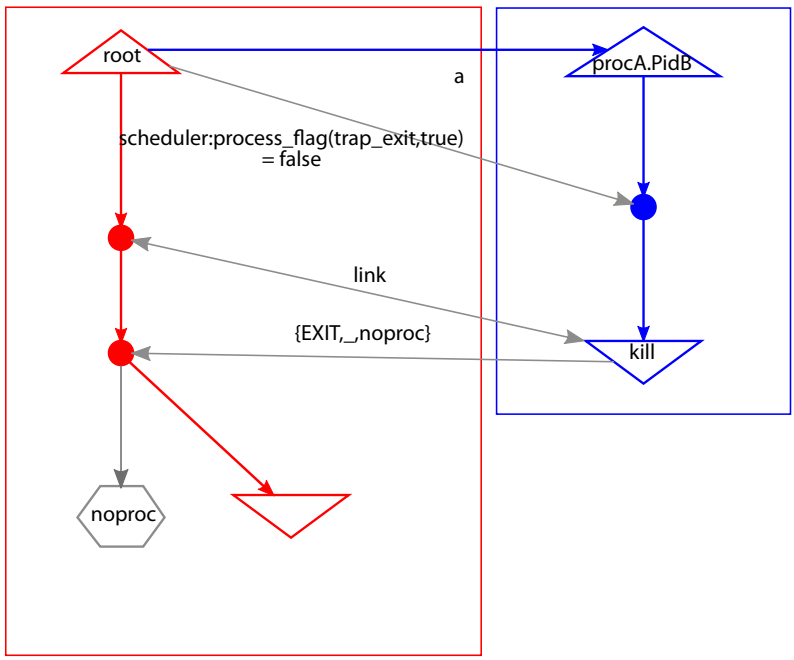


Figure 5.2: An alternative possible execution.

The diagrams help us to understand traces by gathering together all the events that affect one process into one box; in the original traces,

[5]If messages are consumed from a process mailbox out-of-order, then we show the delivery of a message to the mailbox, and its later consumption, as separate transitions.

these events may be scattered throughout the entire trace. But notice that the diagrams also *abstract away* from irrelevant information—specifically, the order in which messages are delivered to *different* processes, which is insignificant in Erlang. This abstraction is one strong reason why the diagrams are easier to understand than the traces they are generated from.

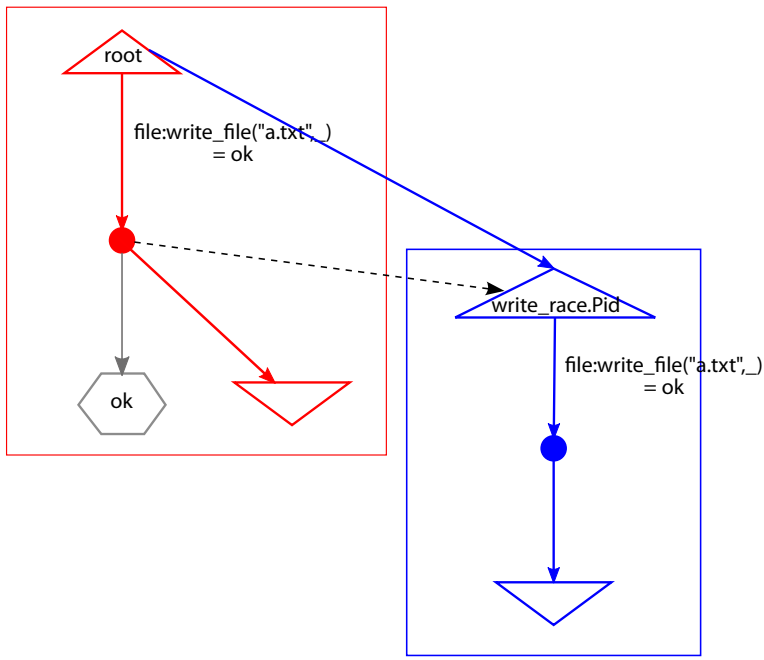


Figure 5.3: A race between two side-effects.

However, we *do* need to know the order in which calls to functions with side-effects occur, even if they are made in different processes. To make this order visible, we add dotted black arrows to our diagrams, from one side-effecting call to the next. Figure 5.3 illustrates one possible execution of this program, in which two processes race to write to the same file:

```
write_race() ->
  Pid = spawn(fun() ->
                file:write_file("a.txt","a")
              end),
  file:write_file("a.txt","b").
```

In this diagram, we can see that the `write_file` in the `root` process preceded the one in the spawned process `write_race.Pid`.

If we draw these arrows between *every* side-effect and its successor, then our diagrams rapidly become very cluttered. However, it is only necessary to indicate the sequencing of side-effects explicitly *if their sequence is not already determined*. For each pair of successive side-effect transitions, we thus compute Lamport's "happens before" relation [Lamport, 1978] between them, and if this already implies that the first precedes the second, then we draw no arrow in the diagram. Interestingly, in our examples then this eliminates the majority of such arrows, and those that remain tend to surround possible race conditions—where the message passing (synchronization) does not enforce a particular order of side-effects. Thus black dotted arrows are often a sign of trouble.

6.1 Analyzing the `proc_reg` race conditions

Interestingly, as we saw in Section 5.5, when we instrumented `proc_reg` and tested it using PULSE and QuickCheck, we obtained a different—even simpler—minimal failing test case, than the one we had previously discovered using QuickCheck with the built-in Erlang scheduler. Since we need to use PULSE in order to obtain a trace to analyze, then we must fix this bug first, and see whether that also fixes the first problem we discovered. The failing test we find using PULSE is this one:

```
{[{set,{var,9},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,10},{call,proc_reg_eqc,kill,[{var,9}]}}],
 {[{set,{var,15},{call,proc_reg_eqc,reg,[c,{var,9}]}}],
  [{set,{var,12},{call,proc_reg_eqc,reg,[c,{var,9}]}}]}}
```

In this test case, we simply create a dead process (by spawning a process and then immediately killing it), and try to register it twice in parallel, and as it happens the first call to `reg` raises an exception. The diagram we generate is too large to include in full, but in Figure 5.4 we reproduce the part showing the problem.

In this diagram fragment, the processes we see are, from left to right, the `proc_reg` server, the second parallel fork (`BPid`), and the first parallel fork (`APid`). We can see that `BPid` first inserted its argument into the ETS table, recording that the name `c` is now taken, then sent an asynchronous message to the server (`{cast,{..}}`) to inform it of the new entry. Thereafter `APid` tried to insert an ETS entry with the same name—but failed. After discovering that the process being registered is actually dead, `APid` sent a message to the server asking it to "audit"

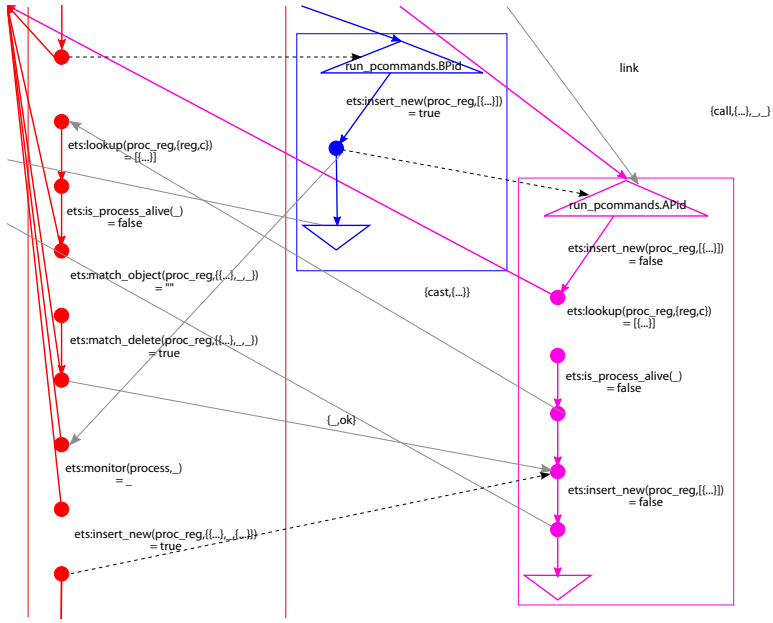


Figure 5.4: A problem caused by message overtaking.

its entry (`{call,{..},_,_}`)—that is, clean up the table by deleting the entry for a dead process. *But this message was delivered before the message from* `BPid`*!* As a result, the server could not find the dead process in its table, and failed to delete the entry created by `BPid`, leading `APid`'s second attempt to create an ETS entry to fail also—which is not expected to happen. When `BPid`'s message is finally received and processed by the server, it is already too late.

The problem arises because, while the clients create "forward" ETS entries linking the registered name to a pid, it is the server which creates a "reverse" entry linking the pid to its monitoring reference (created by the server). It is this reverse entry that is used by the server when asked to remove a dead process from its tables. We corrected the bug by letting clients (atomically) insert *two* ETS entries into the same table: the usual forward entry, and a dummy reverse entry (lacking a monitoring reference) that is later overwritten by the server. This dummy reverse entry enables the server to find and delete both entries in the test case above, thus solving the problem.

In fact, the current Erlang virtual machine happens to deliver messages to local mailboxes instantaneously, which means that one message

cannot actually overtake another message sent earlier—the cause of the problem in this case. This is why this minimal failing test was not discovered when we ran tests on a multi-core, using the built-in scheduler. However, this behavior is not guaranteed by the language definition, and indeed, messages between nodes in a distributed system *can* overtake each other in this way. It is expected that future versions of the virtual machine may allow message overtaking even on a single "many-core" processor; thus we consider it an advantage that our scheduler allows this behavior, and can provoke race conditions that it causes.

It should be noted that exactly the same scenario can be triggered in an alternative way (without parallel processes and multi-core!); namely if the `BPid` above is preempted between its call to `ets:insert_new` and sending the `cast`-message. However, the likelihood for this is almost negligible, since the Erlang scheduler prefers running processes for relatively long time-slices. Using PULSE does not help triggering the scenario in this way either. PULSE is not in control at any point between `ets:insert_new` and sending the `cast`-message, meaning that only the Erlang scheduler controls the execution. Therefore, the only feasible way to repeatedly trigger this faulty scenario is by delaying the `cast`-message by using PULSE (or a similar tool).

6.2 A second race condition in `proc_reg`

Having corrected the bug in `proc_reg` we repeated the QuickCheck test. The property still fails, with the same minimal failing case that we first discovered (which is not so surprising since the problem that we fixed in the previous section cannot actually occur with today's VM). However, we were now able to reproduce the failure with PULSE, as well as the built-in scheduler. As a result, we could now analyze and debug the race condition. The failing case is:

```
{[{set,{var,4},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,7},{call,proc_reg_eqc,kill,[{var,4}]}},
  {set,{var,12},{call,proc_reg_eqc,reg,[b,{var,4}]}}],
 {[{set,{var,18},{call,proc_reg_eqc,reg,[b,{var,4}]}}],
  [{set,{var,21},{call,proc_reg_eqc,reg,[b,{var,4}]}}]}]}
```

In this test case we also create a dead process, but we try to register it once in the sequential prefix, before trying to register it twice in parallel. Once again, one of the calls to `reg` in the parallel branches raised an exception.

Turning again to the generated diagram, which is not included in the paper for space reasons, we observed that both parallel branches (`APid`

and BPid) fail to insert b into the ETS table. They fail since the name b was already registered in the sequential part of the test case, and the server has not yet processed the DOWN message generated by the monitor. Both processes then call where(b) to see if b is *really* registered, which returns undefined since the process is dead. Both APid and BPid then request an "audit" by the server, to clean out the dead process. After the audit, both processes assume that it is now OK to register b, there is a race condition between the two processes, and one of the registrations fails. Since this is not expected, an exception is raised. (Note that if b were alive then this would be a perfectly valid race condition, where one of the two processes successfully registers the name and the other fails, but the specification says that the registration should always return true for dead processes).

This far into our analysis of the error it became clear that it is an altogether rather unwise idea ever to insert a dead process into the process registry. To fix the error we added a simple check that the process is alive before inserting it into the registry. The effect of this change on the performance turned out to be negligible, because is_process_alive is very efficient for local processes. After this change the module passed 20 000 tests, and we were satisfied.

7 Discussion and Related Work

Actually, the "fix" just described does not really remove all possible race conditions. Since the diagrams made us understand the algorithm much better, we can spot another possible race condition: If APid and BPid try to register the same pid at the same time, and that process dies *just after* APid and BPid have checked that it is alive, then the same problem we have just fixed, will arise. The reason that our tests succeeded even so, is that a test must contain *three* parallel branches to provoke the race condition in its new form—two processes making simultaneous attempts to register, and a third process to kill the pid concerned at the right moment. Because our parallel test cases only run *two* concurrent branches, then they can never provoke this behavior.

The best way to fix the last race condition problem in proc_reg would seem to be to simplify its API, by restricting reg so that a process may only register itself. This, at a stroke, eliminates the risk of two processes trying to register the same process at the same time, and guarantees that we can never try to register a dead process. This simplification was actually made in the production version of the code.

	K = 2	K = 3	K = 4	K = 5
N = 1	2	6	24	**120**
N = 2	6	90	**2520**	113400
N = 3	20	**1680**	369600	10^8
N = 4	70	34650	6×10^7	3×10^{11}
N = 5	252	756756	10^{10}	6×10^{14}
⋮	⋮	⋮	⋮	⋮
N = 8	**12870**	10^{10}	10^{17}	8×10^{24}

Figure 5.5: Possible interleavings of parallel branches

Parallelism in test cases

We could, of course, generate test cases with three, four, or even more concurrent branches, to test for this kind of race condition too. The problem is, as we explained in section 4.2, that the number of possible interleavings grows extremely fast with the number of parallel branches. The number of interleavings of K sequences of length N are as presented in Figure 5.5.

The practical consequence is that, if we allow more parallel branches in test cases, then we must restrict the length of each branch correspondingly. The bold entries in the table show the last "feasible" entry in each column—with three parallel branches, we would need to restrict each branch to just three commands; with four branches, we could only allow two; with five or more branches, we could allow only one command per branch. This is in itself a restriction that will make some race conditions impossible to detect. Moreover, with more parallel branches, there will be even more possible schedules for PULSE to explore, so race conditions depending on a precise schedule will be correspondingly harder to find.

There is thus an engineering trade-off to be made here: allowing greater parallelism in test cases may in theory allow more race conditions to be discovered, but in practice may reduce the probability of finding a bug with each test, while at the same time increasing the cost of each test. We decided to prioritize longer sequences over more parallelism in the test case, and so we chose K = 2. However, in the future we plan to experiment with letting QuickCheck randomly choose K and N from the set of feasible combinations. To be clear, note that K only refers to the parallelism in the test case, that is, the number of processes that make calls to the API. The system under test may have hundreds of

processes running, many of them controlled by PULSE, independently of K.

The problem of detecting race conditions is well studied and can be divided in runtime detection, also referred to as *dynamic detection*, and analyzing the source code, so called *static detection*. Most results refer to race conditions in which two threads or processes write to shared memory (data race condition), which in Erlang cannot happen. For us, a race condition appears if there are two schedules of occurring side effects (sending a message, writing to a file, trapping exits, linking to a process, etc) such that in one schedule our model of the system is violated and in the other schedule it is not. Of course, writing to a shared ETS table and writing in shared memory is related, but in our example it is allowed that two processes call ETS insert in parallel. By the atomicity of insert, one will succeed, the other will fail. Thus, there is a valid race condition that we do not want to detect, since it does not lead to a failure. Even in this slightly different setting, known results on race conditions still indicate that we are dealing with a hard problem. For example, Netzer and Miller [1990] show for a number of relations on traces of events that ordering these events on 'could have been a valid execution' is an NP-hard problem (for a shared memory model). Klein et al. [2003] show that statically detecting race conditions is NP-complete if more than one semaphore is used.

Thus, restricting `eqc_par_statem` to execute only two processes in parallel is a pragmatic choice. Three processes may be feasible, but real scalability is not in sight. This pragmatic choice is also supported by recent studies [Lu et al., 2008], where it is concluded that: *"Almost all (96%) of the examined concurrency bugs are guaranteed to manifest if certain partial order between 2 threads is enforced."*

Hierarchical approach

Note that our tools support a *hierarchical* approach to testing larger systems. We test `proc_reg` *under the assumption that the underlying ets operations are atomic*; PULSE does not attempt to (indeed, cannot) interleave the executions of single ETS operations, which are implemented by C code in the virtual machine. Once we have established that the `proc_reg` operations behave atomically, then *we can make the same assumption* about them when testing code that makes use of them. When testing for race conditions in modules that use `proc_reg`, then we need not, and do not want to, test for race conditions in `proc_reg` itself.

As a result, the PULSE schedules remain short, and the simple random scheduling that we use suffices to find schedules that cause failures.

Model Checking

One could argue that the optimal solution to finding race conditions problem would be to use a model checker to explore all possible interleavings. The usual objections are nevertheless valid, and the rapidly growing state space for concurrent systems makes model checking totally infeasible, even with a model checker optimized for Erlang programs, such as McErlang [Fredlund and Svensson, 2007]. Further it is not obvious what would be the property to model check, since the atomicity violations that we search for can not be directly translated into an LTL model checking property.

Input non-determinism

PULSE provides deterministic scheduling. However, in order for tests to be repeatable we also need the external functions to behave consistently across repeated runs. While marking them as side-effects will ensure that they are only called serially, PULSE does nothing to guarantee that functions called in the same sequence will return the same values in different runs. The user still has to make sure that the state of the system is reset properly before each run. Note that the same arguments apply to QuickCheck testing; it is crucial for shrinking and re-testing that input is deterministic and thus it works well to combine QuickCheck and PULSE.

False positives

In contrast to many race finding methods, that try to spot common patterns leading to concurrency bugs, our approach does not produce false positives and not even does it show races that result in correct execution of the program. This is because we employ *property-based testing* and classify test cases based on whether the results satisfy correctness properties and report a bug only when a property is violated.

Related tools

Park and Sen [2008] study atomicity in Java. Their approach is similar to ours in that they use a random scheduler both for repeatability and increased probability of finding atomicity violations. However, since

Java communication is done with shared objects and locks, the analysis is rather different.

It is quite surprising that our simple randomized scheduler—and even just running tests on a multi-core—coupled with repetition of tests to reduce non-determinism, should work so well for us. After all, this can only work if the probability of provoking the race condition in each test that contains one is reasonably high. In contrast, race conditions are often regarded as very *hard* to provoke because they occur so rarely. For example, Sen used very carefully constructed schedules to provoke race conditions in Java programs [Sen, 2008]—so how can we reasonably expect to find them just by running the same test a few times on a multi-core?

We believe two factors make our simple approach to race detection feasible.

- Firstly, Erlang is not Java. While there *is* shared data in Erlang programs, there is much less of it than in a concurrent Java program. Thus there are many fewer potential race conditions, and a simpler approach suffices to find them.

- Secondly, we are searching for race conditions during *unit testing*, where each test runs for a short time using only a relatively small amount of code. During such short runs, there is a fair chance of provoking race conditions with any schedule. Finding race conditions during whole-program testing is a much harder problem.

Chess, developed by Musuvathi et al. [2008], is a system that shares many similarities with PULSE. Its main component is a scheduler capable of running the program deterministically and replaying schedules. The key difference between Chess and PULSE is that the former attempts to do an exhaustive search and enumerate all the possible schedules instead of randomly probing them. Several interesting techniques are employed, including prioritizing schedules that are more likely to trigger bugs, making sure that only fair schedules are enumerated and avoiding exercising schedules that differ insignificantly from already visited ones.

Visualization

Visualization is a common technique used to aid understanding software. Information is extracted statically from source code or dynamically from execution and displayed in graphical form. Of many software visualization tools a number are related to our work. Topol et al. [1995]

developed a tool that visualizes executions of parallel programs and shows, among other things, a trace of messages sent between processes indicating the happened-before relationships. Work of Jerding et al. [1997] is able to show dynamic call-graphs of object-oriented programs and interaction patterns between their components. Arts and Fredlund [2002] describe a tool that visualizes traces of Erlang programs in form of abstract state transition diagrams. Artho et al. [2007] develop a notation that extends UML diagrams to also show traces of concurrent executions of threads, Maoz et al. [2007] create event sequence charts that can express which events "must happen" in all possible scenarios.

8 Conclusions

Concurrent code is hard to debug and therefore hard to get correct. In this paper we present an extension to QuickCheck, a user level scheduler for Erlang (PULSE), and a tool for visualizing concurrent executions that together help in debugging concurrent programs. The tools allow us to find concurrency errors on a module testing level, whereas industrial experience is that most of them slip through to system level testing, because the standard scheduler is deterministic, but behaves differently in different timing contexts.

We contributed `eqc_par_statem`, an extension of the state machine library for QuickCheck that enables parallel execution of a sequence of commands. We generate a sequential prefix to bring the system into a certain state and continue with parallel execution of a suffix of independent commands. As a result we can provoke concurrency errors and at the same time get good shrinking behavior from the test cases.

We contributed with PULSE, a user level scheduler that enables scheduling of any concurrent Erlang program in such a way that an execution can be repeated deterministically. By randomly choosing different schedules, we are able to explore more execution paths than without such a scheduler. In combination with QuickCheck we get in addition an even better shrinking behavior, because of the repeatability of test cases.

We contributed with a graph visualization method and tool that enabled us to analyze concurrency faults more easily than when we had to stare at the produced traces. The visualization tool depends on the output produced by PULSE, but the use of computing the "happens before" relation to simplify the graph is a general principle.

We evaluated the tools on a real industrial case study and we detected

two race conditions. The first one by only using `eqc_par_statem`; the fault had been noticed before, but now we did not need to instrument the code under test with `yield()` commands. The first and second race condition could easily be provoked by using PULSE. The traces recorded by PULSE were visualized and helped us in clearly identifying the sources of the two race conditions. By analyzing the graphs we could even identify a third possible race condition, which we could provoke if we allowed three instead of two parallel processes in `eqc_par_statem`.

Our contributions help Erlang software developers to get their concurrent code right and enables them to ship technologically more advanced solutions. Products that otherwise might have remained a prototype, because they were neither fully understood nor tested enough, can now make it into production. The tool PULSE and the visualization tool are available under the Simplified BSD License and have a commercially supported version as part of Quviq QuickCheck.

Acknowledgements

This research was sponsored by EU FP7 Collaborative project *ProTest*, grant number 215868.

6

Accelerating Race Condition Detection through Procrastination

This paper was published at the Erlang workshop 2011 in Tokyo.

Chapter 6

Accelerating Race Condition Detection through Procrastination

Thomas Arts John Hughes Ulf Norell
Nicholas Smallbone Hans Svensson

Abstract

Race conditions are notoriously frustrating to find, and good tools can help. The main difficulty is *reliably* provoking the race condition. In previous work we presented a *randomising scheduler* for Erlang that helps with this task.

In a language without pervasive shared mutable state, such as Erlang, performing scheduling decisions at random uncovers race conditions surprisingly well. However, it is not always enough. We describe a technique, *procrastination*, that aims to provoke race conditions more often than by random scheduling alone. It works by running the program and looking for pairs of events that might interfere, such as two message sends to the same process. Having found such a pair of events, we re-run the program but try to provoke a race condition by reversing the order of the two events.

We apply our technique to a piece of industrial Erlang code. Compared to random scheduling alone, procrastination allows us to find minimal failing test cases more reliably and more quickly.

1 Introduction

Now that multicore processors are ubiquitous, concurrent programming has become as inescapable as it is difficult. The Erlang programming language [Armstrong, 2007] was designed to make concurrent programming easy, by making common concurrency mistakes impossible. Erlang processes do not share memory, and thus cannot corrupt each others' data. Erlang data structures are immutable, and thus can be freely copied between process heaps, or between distributed nodes. Erlang processes communicate and synchronize by passing immutable messages from one process to another. These design decisions make *data races*, the scourge of concurrent imperative programming, absolutely impossible.

Nevertheless, Erlang programmers make plenty of concurrency errors. The order of message delivery can vary, leading to scheduling-dependent behaviour. Erlang processes can access global tables managed by the virtual machine—admittedly via an atomic API—but table entries can be used as global variables to recreate the same kind of data races found in other languages. The file store also represents a global state that can lead to race conditions between processes. Erlang's design ameliorates, but does not eliminate race conditions—and thus, tools for race condition testing are still highly relevant.

In Paper 5, we implemented a randomizing scheduler for Erlang, and used it to find race conditions in industrial Erlang code. While data races take only a limited number of forms [Vaziri et al., 2006], races in message-passing programs are harder to characterize, and so we detected races in a *black box* manner, as violations in *serializability* of an API under test. We combined our scheduler with a random testing tool, QuickCheck [Arts et al., 2006], which generates and simplifies random tests containing sequential and parallel invocations of the API under test. In this way we were able to find *minimal test cases* that exhibit serializability violations, and from them diagnose the underlying race conditions with relative ease.

The search for minimal failing test cases proceeds in two phases: first we search for *any* failing test, then having found one, we search for *simplifications* of that test that also fail. In both phases, we need to determine whether a candidate test case can provoke a race, and in our previous work we did so by running each test many times with different random schedules. If the race we are looking for occurs only rarely, then we may need to run each test case very many times to determine with reasonable accuracy whether or not the race is present. Failure

to provoke the race may lead either to failure to find a failing test at all (in the first phase), or failure to simplify the failed test to a minimal example. In our case study, we found we needed to run each smaller test around 10–100 times to obtain good results. This makes testing quite slow.

Finding *data races* in imperative programs by random scheduling is much more difficult, because we must schedule each memory access to shared data, rather than larger scale atomic operations such as message delivery or table access. This motivated Sen to introduce *race directed* random testing [Sen, 2008], in which a test is run once using a random schedule, *possible races* are identified from this first run, and the test is then run again many times using schedules which are specially constructed to provoke each possible race. In Sen's case, a possible race consists of two memory accesses A and B to the same location in different threads, whose execution potentially could be reversed, and the specially constructed schedule tries to delay A until after the execution of B. We call this process of delaying A "*procrastination*". While Sen's paper discusses delaying memory accesses, it is clear that the same idea can be applied to the schedules we construct for Erlang programs—and that, by improving the probability of provoking a race, it could enable us to find minimal failing test cases much more quickly.

The contributions we present in this paper are

- We transferred to Erlang the idea of using potentially conflicting actions to guide a search for race conditions, which is explained in Section 3.
- Validation of procrastination on an industrial example (Section 4). We show that procrastination is effective and that first-order procrastination seems more useful than higher-order procrastination for this example.
- We extended PULSE to be able to run infeasible schedules (Section 3), in order to re-use as much as possible from a given schedule. This saves a lot of expensive analysis that other approaches need to deal with.
- We experimented with procrastination while shrinking test cases (Section 6) and noticed that in particular the ability to re-use a schedule was very effective (Section 6.3).

In Section 2 we provide background on QuickCheck and PULSE, the framework to which we added procrastination. In Section 7 we

compare our approach with other approaches to race detection in different contexts.

2 Background

QuickCheck

QuickCheck [Claessen and Hughes, 2000] is a tool that tests universally quantified *properties*, instead of single test cases. QuickCheck generates random test cases from each property, tests whether the property is true in each case, and reports cases for which the property fails. Recent versions also "shrink" failing test cases automatically, by searching for similar, but smaller test cases that also fail. The result of shrinking is a "minimal"[1] failing case, which often makes the root cause of the problem very easy to find.

Quviq QuickCheck is a commercial version that includes support for model-based testing using a state machine model [Hughes, 2007]. It has been used to test a wide variety of industrial software, such as Ericsson's Media Proxy [Arts et al., 2006] among others. State machine models define a set of API calls to include in test cases, pre- and post-conditions for each call, and the corresponding state transitions on a model state. QuickCheck then generates well-formed call sequences (satisfying all preconditions, for example), executes them, and checks postconditions with respect to the model state.

Although these state machine models specify the *sequential* behaviour of the API under test, surprisingly, they can also be used to test for race conditions! Assuming that the API calls are intended to behave atomically, then we can generate parallel call sequences and adjudge the test by determining whether there is any *interleaving* of the calls that can explain the actual results observed. Our parallel test cases consist of a random sequential prefix to put the system into a random state, followed by two or more random call sequences executed in parallel. If there is no serialization of the test case that satisfies the postconditions in the model, then the test fails. Details of the method can be found in Chapter 5.

Note that this is a form of *black box* race condition testing: we are only interested in *races that cause a violation of the postconditions*, not in races that lead to non-deterministic, but still *valid* results. Also, note that the approach is only applicable to APIs that are intended to

[1] Minimal in the sense that none of the similar, smaller tests failed.

behave atomically. This is of course a limitation, but very many APIs do have this property (or at least, important parts of them do), and the payoff is that very little extra work is required to reuse a sequential state machine model for parallel testing too. (In many cases, it simply requires changing a call of `commands` to a call of `parallel_commands`.)

Simplest is to execute the parallel tests using native Erlang concurrency, relying on the inherent non-determinism of execution on a multicore processor to provoke races. However, because of determinism in the native Erlang scheduler—and perhaps in the processor itself—finding races in this manner can be slow. To speed up their detection, we implemented our own randomizing scheduler, PULSE.

PULSE

PULSE is a *user-level scheduler*, sitting on top of the normal Erlang scheduler. Its aim is to take control over the sources of non-determinism in Erlang programs introduced by scheduling decisions. This means that we can introduce more randomness in schedules, but also that we can repeat a test using exactly the same schedule by simply recording the scheduling decisions: this makes tests repeatable.

Since PULSE is a user-level scheduler, to give it control over a piece of code the code must be *instrumented* to co-operate with PULSE. Rather than forcing the user to instrument their code themselves, we have a `parse_transform` which does this automatically. We also provide a macro that takes care of running a piece of code under PULSE and collecting the results, so that it takes a reasonably small amount of effort to use PULSE during testing.

The central idea is to provide absolute control over the order of relevant events. Relevant events are interaction of a process with its environment, so called *side-effects*. Code instrumentation replaces each call to side-effect containing functions by a function that gives control to PULSE. Of particular interest in Erlang is the way processes interact by message passing, which is asynchronous. Message channels, containing messages that have been sent but not yet delivered, are thus part of the environment and explicitly modeled as such in PULSE. It makes sense to separate side-effects into two kinds: *outward* side-effects, that influence only the environment (such as sending a message over a channel, which does not block and cannot fail, or printing a message), and *inward* side-effects, that allow the environment to influence the behaviour of the process (such as receiving a message from a channel, or asking for the system time).

PULSE controls its processes by allowing only one of them to run at a time. It employs a cooperative scheduling method: At each decision point, PULSE randomly picks one of its waiting processes to proceed, and wakes it up. The process may now perform a number of outward side-effects, which are all recorded and taken care of by PULSE, until the process wants to perform an inward side-effect. At this point, the process is put back into the set of waiting processes, and a new decision point is reached.

In addition to sending and receiving messages between themselves, the processes under test can also interact with uninstrumented code. PULSE then controls the order in which those interactions take place. We allow the programmer to specify which external functions have side-effects. Each call of a side-effecting function is then instrumented with code that `yield`s before performing the real call, allowing PULSE to run another process at that point.

Side-effecting functions are treated as atomic which is also an important feature that aids in testing systems built of multiple components. Once we establish that a component contains no race conditions we can remove the instrumentation from it and mark its operations as atomic side-effects. We will then be able to test other components that use it and each operation marked as side-effecting will show up as a single event in a trace. Therefore, it is possible to test a component for race conditions independently of the components that it relies on.

A Problem!

As we showed in Chapter 5, randomized scheduling worked well, almost surprisingly well. However, there are still many situations where a completely random schedule fails to expose an error, and where steering the scheduler can improve the search efficiency. To illustrate the problem, consider the example in Fig. 6.1. In the example we spawn a process `A`, that first sends the message `hello` directly to a process `C`, and then sends the message `world` to `B`, which forwards it through a chain of `100` proxy processes to `C`. In this (contrived) example there is an obvious race between the two messages, i.e. whether C first receives `hello` or `world`. Still, in practice, regardless of whether the normal Erlang scheduler or PULSE is used, `hello` will never[2] arrive after `world`. If we use PULSE to schedule a run of the example we have to choose, `100` times in a row,

[2] As in: not in our lifetime!

```
example() ->
  C = self(),
  B = proxy(100, C),
  A = spawn(fun() ->
              C ! hello,
              B ! world
            end),
  receive Msg1 -> ok end,
  receive Msg2 -> ok end,
  {Msg1, Msg2}.

proxy(0, Pid) ->
  Pid;
proxy(N, Pid) ->
  Proxy = proxy(N-1, Pid),
  proxy(Proxy).

proxy(Pid) ->
  spawn(fun() ->
          receive Msg -> Pid ! Msg end
        end).
```

Figure 6.1: World-Hello with Proxy

to forward the `world` message instead of delivering `hello`. This will happen in 1 out of 2^{100} cases.

3 Procrastination

The main idea behind the procrastination technique is simple: identify potentially conflicting actions, and use the potential conflicts to steer the scheduler. The idea is inspired by race-condition testing of C and Java programs [Lai et al., 2010, Savage et al., 1997], where conflicting memory accesses are recorded and used to steer scheduling.

PULSE records all scheduling decisions taken during program execution. Below is an excerpt from a schedule obtained running the "world hello" example. Here the only side-effect is message delivery: `{A,{deliver,B}}` means that a message from `B` was delivered to `A`. Other side-effects would show up as `yields`, where `{A,yield,{Fun,Args}}`

means that process A performed the side-effecting function Fun (with arguments Args).

```
[{root-proxy99, {deliver, root-example.A }},
 {root,         {deliver, root-example.A }},
 {root-proxy98, {deliver, root-proxy99   }},
 {root-proxy97, {deliver, root-proxy98   }},
 ...
 {root,         {deliver, root-proxy     }}]
```

Given a schedule we can identify potential conflicts. For message delivery there is a potential conflict whenever a process receives messages from two other distinct processes. In the case of user defined side-effects, we rely on the user to define which operations potentially conflict, via a simple call-back function.

As an example consider the fictitious schedule in which process A receives a message from both B and C:

```
[{process_A, {deliver, process_B}},
 {process_B, yield, {do_it,[1,2,3]}},
 ...
 {process_A, {deliver, process_C}},
 ...]
```

Our algorithm detects a potential conflict between the two deliver actions in the schedule. To use this conflict to steer PULSE we create the re-ordered schedule:

```
[{process_B, yield, {do_it,[1,2,3]}},
 ...
 {process_A, {deliver, process_C}},
 {process_A, {deliver, process_B}},
 ...]
```

By feeding the re-ordered schedule to PULSE we delay (procrastinate!) the delivery from process_B until after the delivery from process_C. However, there is a caveat, namely that the re-ordered schedule might not be feasible. For instance, it might be that the message from process_C is a response to a request that process_A makes after receiving the message from process_B. In this case, when running the re-ordered schedule, we will get to the point where we're supposed to deliver the message from process_C but there will be no such message waiting to be delivered.

There are two main solutions to the problem with infeasible schedules: (1) make the analysis more exact to avoid creating infeasible schedules, or (2) allow the scheduler to follow infeasible schedules in some way. We opted for the second solution, and adapted PULSE accordingly. The new version of PULSE tries to follow the given schedule, but whenever there is a scheduling decision that is infeasible (such as delivering a message that has not in fact been sent—such decisions are easy to detect at runtime), it is discarded and PULSE tries the next action in the schedule. With this relaxed strategy PULSE might run out of schedule to follow—if this happens then it reverts to its original, purely random, strategy. Interestingly, in most of the related work, the opposite approach is taken, and various techniques such as "happens before"-relations are used, e.g. [Lai et al., 2010]. There is good reason for this extra analysis, since an infeasible schedule in a normal scheduler (for example the Java VM) might result in a dead-lock of the whole system. Here, since we have full control of the scheduler, we are in a better position and can take the simpler path without expensive analysis.

It is easy to detect at runtime when a scheduling decision is infeasible: we simply look at the set of actions that we would've chosen from had we been following a random schedule, and check that the action we are about to take is in that set. This also makes it impossible for procrastination to behave *incorrectly*—i.e. to fail to respect the semantics of Erlang—since the behaviours that we provoke are ones that our purely random scheduler could also have provoked, given enough luck.

3.1 Procrastinating World Hello

What happens now if we apply procrastination to the "world hello" example? We have a schedule from PULSE as shown before where there is actually only one potential conflict, namely between the delivery from `root-example.A` and the delivery from `root-proxy` to `root`. (Where `root` is `C` in the example, `root-example.A` is `A` and `root-proxy` is the last process in the proxy chain.) Our re-ordered schedule would then be:

```
[{root-proxy99, {deliver, root-example.A }},
 {root-proxy98, {deliver, root-proxy99   }},
 ...
 {root,         {deliver, root-proxy     }},
 {root,         {deliver, root-example.A }}]
```

If we supply this schedule to PULSE the result of running the program is what we tried to achieve, namely `{world,hello}`. (This process is normally automatic, we show the concrete steps as an explanation.)

```
27> pulse:run(fun() -> example() end,
              [{schedule,ReOrderedSchedule}]).
...
{world,hello}
```

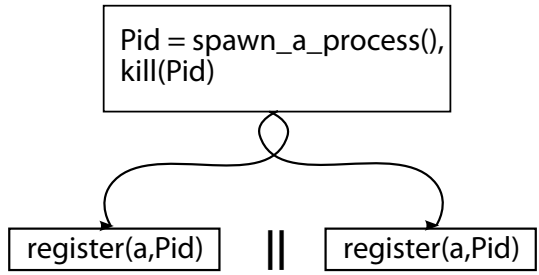
4 The ProcReg Example

Procrastination works well in the simple "world hello" example—but how does it perform in a more realistic setting? To investigate this question, we applied it to the industrial case study used in Chapter 5—the proc_reg *process registry*.

In Erlang, every process has a unique, dynamically assigned process identifier (*pid*); the only way to communicate with a process is via its pid. To enable processes to find each other's pids, Erlang provides a *process registry*—a kind of local name server—which associates names with pids. The registry provides an API with operations to register a process with a name, look up the pid associated with a name, and unregister a name. It is heavily used to provide access to system services: a newly started Erlang node already contains thirteen registered processes.

However, the built-in registry does impose sometimes-unwanted restrictions. Names are restricted to be Erlang *atoms*, rather than more general terms. No process may simultaneously be registered with two different names. To lift these restrictions, Wiger developed an extended process registry in Erlang—much easier to modify than the one built into the virtual machine. Wiger's code has been in use in Ericsson products for several years [Wiger, 2007].

Our case study is a prototype version of Wiger's registry, with a known race condition. This prototype consists of a registry server that responds to client requests and stores registration data in a so-called *ets table*—hash tables built in to the virtual machine. The use of a single server process serializes operations on the ets table. But, as an optimization, the prototype performs some ets table operations on the client side—thus introducing the possibility of a race condition. And indeed, a race condition occurs. One case that can provoke it is illustrated by this diagram:



Here a process is created and killed, resulting in a dead process, registered with the name `a`, then registered twice in parallel under the same name. The intention is that the registry contain only *live* processes, and that requests to register a dead pid are ignored. For consistency with the built-in registry, registering a dead pid should return `true`. But in this example, one of the parallel calls of `register` occasionally raises an exception instead.

Our QuickCheck specification of the registry models the state as a set of alive pids, a set of dead pids, and a set of name/pid pairs. The specification also defines transition functions on this state for each operation, and checks postconditions against this model state. In this example, the model tells us that `Pid` is a dead process, and therefore the postcondition for `register` requires the result to be `true`. We adjudge parallel test cases like this one by determining whether there is any serialization of the test case in which all the postconditions hold. In this example there are only two serializations, and in both of them, all the calls of `register` should return `true`. When an exception is raised instead, then our QuickCheck property fails.

In fact, this prototype was abandoned in 2005, after QuickCheck revealed the *existence* of a race condition, but we were unable to diagnose it. At that time we could only generate very large failing test cases. It was not until 2009 that we were able to *shrink* the failing test cases to the example above, and with its help, find and correct the error in the code. But now we know what the bug is; in this paper our focus is on the *performance* of our testing.

The QuickCheck properties that we use to test the process registry generate sequences of operations to spawn, kill, register, or unregister a process, or look up a name, all with equal probability. We generate parallel tests by splitting the tail end of the sequence into two parallel sequences (respecting preconditions). We then *run* these parallel tests in a variety of different ways.

4.1 Measurements

The simplest approach is just to run the tests using the built-in Erlang scheduler on a multicore computer (we used an eight core i7 machine with 16GB RAM for all our tests). That is, we relied on the non-determinism inherent in parallel execution to provoke race conditions. When we did so, we found that slightly less than 0.1% of tests failed, which enabled us to find 59 failing cases in 30 minutes of testing (or a mean-time-to-failure of 30.5s). Note that since the known race condition depends on trying to register a dead process twice in parallel, then many generated test cases cannot fail at all. For this particular bug, around 8% of the generated test cases can fail. On the other hand, even a test case that potentially provokes the race will pass in many cases. We found that the failing tests we found in this way, relying solely on parallel execution, failed in approximately one of every 30 runs.

When we ran the same tests using PULSE, our randomizing scheduler, then we saw 741 failures in 30 minutes—more than twelve times as many (mean-time-to-failure 2.43s, c.f. Figure 6.2). This is despite the overheads that PULSE imposes—each test ran 1.5 times slower under PULSE than with the native scheduler. This demonstrates the benefits of randomising the schedule very clearly.

5 Procrastinating ProcReg

We need to refine the ideas of Section 3 a little in order to be able to cope with real-world Erlang code such as ProcReg.

Dealing with side effects Unlike the "world hello" example, the ProcReg example contains side effects other than message passing, namely reads and writes to the Erlang mutable term storage, *ets*.

As described in Section 2, PULSE can deal with arbitrary side-effecting operations: we may register the ets functions as having a side effect. Thereafter, whenever a process wants to read or write to an ets table, it will first ask PULSE for permission; at this point PULSE might delay the execution of the process arbitrarily.[3]

These side effects are recorded as decisions in the schedule that PULSE produces. Thus, if our program manipulates ets tables then we will get entries such as

[3] This treatment assumes that side effects are atomic, because we never attempt to execute two side effecting operations in parallel—a perfectly reasonable assumption in our case, but one which does not make sense for traditional shared state concurrency.

```
{process_A, yield, {ets, insert, [...]}}
{process_B, yield, {ets, lookup, [...]}}
```

in our schedule. Each entry represents the point when a process was *given permission* to execute a particular side effect. We can apply procrastination to side effecting functions in *exactly the same way* that we do with message deliveries: given a schedule, we identify pairs of function calls that might conflict (such as an `ets:insert` and an `ets:lookup`), and attempt to delay the execution of the first function call until after the execution of the second.

Not all pairs of function calls can conflict, and it is not useful to try to swap the order of two function calls that do not conflict. We allow the user to specify which function calls can conflict: this reduces the total number of procrastinations we try by ruling out some procrastinations that can never be fruitful. PULSE will assume that any two side-effecting functions can conflict if not specified otherwise: this is safe, but by giving more fine-grained conflict information the user can reduce the number of fruitless procrastinations that are tried. In the case of ets, we say that two operations conflict if they operate on the same ets table and at least one of them is a write.

Implementing procrastination Our implementation of procrastination consists of four steps: (1) run the test case with a random schedule using PULSE, (2) for each scheduling decision identify later potentially conflicting decisions, (3) for each scheduling decision try, by running a modified schedule with PULSE, to move it past the *last* potentially conflicting decision, using the techniques of section 3 to detect infeasible schedules, and record which potentially conflicting decisions we managed to move it past (we call these pairs of potentially conflicting schedule decisions *feasible procrastinations*), and (4) for each such feasible procrastination, construct and execute the procrastinated schedule.

Controlling the amount of procrastination There may be many possible ways to procrastinate a given schedule. Thus, for a given test case QuickCheck might spend a considerable amount of time repeating the test case, trying various procrastinations for that test case. However, if the test case is one that cannot provoke a bug then this will get us nowhere! So there is a trade-off between exploring many procrastinations of a single test case and trying many test cases. Therefore we allow the user to choose how many of the feasible procrastinations to try for each schedule.

Scheduler		Time (s)
VM		30.5
PULSE		2.43
Procrastination Level	Limit	
1	$\langle 10 \rangle$	1.32
1	$\langle \infty \rangle$	3.01
2	$\langle 5, 1 \rangle$	3.90
2	$\langle 10, 3 \rangle$	6.47
2	$\langle 10, 10 \rangle$	8.91
2	$\langle 20, 20 \rangle$	16.1

Figure 6.2: Mean time to failure

Higher-order procrastination In our basic approach we find two conflicting scheduling decisions and try to reverse their order. However, it might in general be necessary to procrastinate several decisions at once—especially if the schedule is large. We call this *higher-order* procrastination. Since procrastinating one decision may make other procrastinations possible or impossible, it does not really make sense to apply several simultaneous procrastinations to one schedule. Instead, we first apply one procrastination and re-run the test case to get a *new* schedule. If that procrastination succeeded we can then apply further levels of procrastination to the *new* schedule.

Since the number of procrastinated schedules is exponential in the number of levels of procrastination we try, this technique is not always useful: we allow the user to say if we should apply it or not and how many levels of procrastination to try.

5.1 Measurements

We re-ran the experiment of section 4.1—testing ProcReg and recording how many failing test cases QuickCheck was able to find in 30 minutes—but using procrastination. The results are shown in Figure 6.2.

The "Level" column indicates how many levels of procrastination we tried (see paragraph "Higher-order procrastination") and the "Limit" column shows how many procrastinated schedules we tried at each

level (see paragraph "Controlling the amount of procrastination"), ∞ meaning that we try all possible procrastinations. "Time" is, as before, the mean time to failure.

We can see that procrastination almost doubled the number of failures we were able to provoke in 30 minutes, compared to using random scheduling alone—a reasonable success. However, higher-order procrastination actually made things *worse* than not procrastinating at all. This is because we repeat each test case over and over again even if that test case simply cannot fail—too much procrastination is a bad thing. According to the table above, for ProcReg the happy medium is one level of procrastination with ten decisions from each schedule chosen for procrastination.

Measuring failure reproduction The results above may be deceptive because they take no notice of *which* race conditions each testing method is able to provoke. If a method only works well with simple test cases, it will still get a low mean time to failure if it runs each test quickly—even if it is completely unable to provoke race conditions in more complicated cases.

Therefore, we also tried using each testing method to reproduce bugs that were found by the *other* testing methods. We collected a variety of bug-provoking test cases that were found by each testing method, and for each test case, we repeatedly ran each method on that test case until it reproduced the bug, recording how many attempts we needed to find the bug in each case and how long it took.

The results are shown in figure 6.3. The "Origin" column shows which method we used to originally find the bug and "Scheduler" shows which method we were using to try to reproduce the bug. We list the average amount of time taken to make each test case fail and the average number of times we had to repeat each test case before it failed.

Using the virtual machine's scheduler performs very poorly, as we expect. Even on test cases that were originally found without the help of PULSE, we have a very low probability—about $\frac{1}{100}$ on average—of reproducing the bug when we run the test case. For test cases that were found by PULSE, we need to repeat each test case thousands of times, and because of that, using the virtual machine increases the time taken to reproduce the bugs by a factor of 100.

PULSE is easily able to reproduce bugs that were found using the VM's scheduler, even without procrastination—and here procrastination slows things down by a factor of two. This is because we can only find

Origin	Scheduler	Time (s)	Avg. #tests
VM	VM	0.82	101.3
	PULSE	0.14	2.0
	Procrastination	0.28	1.0
PULSE	VM	20.16	3584.1
	PULSE	0.24	13.6
	Procrastination	0.36	1.16
Procrastination	VM	35.03	7126.6
	PULSE	0.83	70.9
	Procrastination	0.32	1.54

Figure 6.3: Failure detection—grouped by origin

very small counterexamples using the VM's scheduler and for those random scheduling is enough.

For bugs that were found by PULSE without the help of procrastination, it might *seem* from figure 6.3 as though using procrastination greatly reduces the number of tests we need to run. However, using procrastination, each "test" executes the program several times—once for each procrastinated schedule—so the number of tests is a deceptive measure. Looking at the time taken shows that in reality PULSE without procrastination wins here, too.

However, on bugs that we originally found with the help of procrastination, PULSE with procrastination is the surefire winner. We see that for those test cases, using procrastination is more than twice as fast as not using it, and more than 100 times faster than using the VM's scheduler. Since each test using procrastination executes the test case at most 11 times (we are using a limit of 10 procrastinations), we can see that using procrastination reduced the number of times we had to execute the program by a factor of at least 3.

Thus, procrastination really is better at provoking complex race conditions than random scheduling alone. There is one test case in particular for which PULSE without procrastination has great trouble. It is short—7 commands long, and the test case itself is completely sequential—but PULSE without procrastination takes on the order of 1000 attempts to find it on average. Using procrastination—even first-order procrastination with a limit of 10 procrastinated schedules—we find the bug first time, every time. We conjecture that in more complex systems—where

there are more scheduling decisions to be made than in our ProcReg example—there are many such bugs that require extremely good luck to find using random testing but where adding a small amount of procrastination can reliably provoke the error. In particular, with a purely random scheduler, once a particular action is *possible* then it is likely to be chosen within a few steps, since at each step there is a reasonable probability that the scheduler will choose that action. Thus an action will not be delayed for very long in a purely random scheduler; this is what happens in the "world hello" example of section 2, where to provoke an error we need to delay a message delivery for 100 steps and have only a one in 2^{100} chance of doing that. With procrastination, we can delay some decisions for an arbitrary amount of time.

6 Shrinking Counterexamples

In order to understand an error it is important that the failing test case is reasonably small. In the case of the process registry example it was not until we were able to shrink the failing test cases to the one shown in Section 4 that the error could be successfully diagnosed. To achieve this QuickCheck, having found one failing test case, then shrinks it by trying many smaller, similar tests. If any of these fail, the original failing test is replaced by the smaller one and the shrinking process continues. When no smaller tests fail, then a "minimal" failing case has been found.

Since the outcome of a test is non-deterministic, it might be that a test succeeds even though the test case is one that sometimes provokes a bug in our program, if when we run the test we are unlucky with the scheduling decisions we choose. If this happens for all of the candidate shrunk test cases, the shrinking process stops with a result that is not necessarily minimal. Thus it is particularly important during shrinking that test cases that contain the race condition reliably fail when they are executed.

6.1 Improving shrinking by repeating tests

One method for improving the reliability of shrinking is to run each test N times during the shrinking process. If the test fails in any one of these runs it is considered to be a failing case. This increases the likelihood of successfully identifying test cases that contain the race condition at the expense of making shrinking more time consuming. This is the method we used originally to find the minimal failing case for the process registry.

Scheduler	N	Time (s)	Avg. Len.	Max. Len.	Minimal
VM	1	1.74	11.4	45	0.0%
	10	2.68	10.4	45	0.5%
	25	4.19	9.7	47	2.2%
	100	29.04	11.5	48	1.0%
PULSE	1	2.42	12.0	38	0.7%
	10	3.20	7.1	26	17%
	25	5.14	5.9	28	43%
	50	8.22	5.2	27	67%
	100	14.28	4.6	9	90%

Figure 6.4: Shrinking performance when repeating each test N times.

To measure the performance of shrinking we ran the 45 random test cases we collected in which the race condition is present, varying the scheduler (the Erlang virtual machine scheduler or PULSE) and the value of N. For each execution we measured the length of the shrunk test case, the time taken by the shrinking process, and whether or not the shrunk test case was minimal[4]. We measured this 10 times to reduce the impact of coincidentally getting the best performance. Figure 6.4 shows the results for each scheduler and value of N.

The first observation we can make is that when running with PULSE, increasing N improves the quality of the shrinking as we expect, but when running with the Erlang scheduler it has hardly any effect. This is likely due to the Erlang scheduler being quite deterministic and not gratuitously delaying actions, so if a test case does not reveal a race condition the first run it is unlikely to do so in later runs.

Another interesting observation is that going from $N = 1$ to $N = 10$ does not affect the shrinking time significantly. This can be explained by the fact that when shrinking, QuickCheck first tries to take big shrinking steps, throwing away much of the original test case, and only if that fails tries smaller steps like taking out a single command. By repeating each test case during shrinking, QuickCheck is more likely to provoke an error with the large shrinking steps, which reduces the number of steps required to reach a minimal test case.

[4] We deem a test case minimal if we could not provoke a bug in any of its shrinkings within 50 tests using any of the methods at our disposal.

Level	Limit	Time (s)	Avg. Len.	Max. Len.	Minimal
1	⟨10⟩	2.25	5.6	27	60%
1	⟨∞⟩	3.80	5.6	25	56%
2	⟨5, 1⟩	7.01	5.3	15	66%
2	⟨10, 3⟩	8.93	5.2	14	65%
2	⟨10, 10⟩	10.54	5.2	14	66%
2	⟨20, 20⟩	19.36	5.3	13	63%

Figure 6.5: Shrinking performance with procrastination.

6.2 Shrinking with procrastination

Procrastination works in a similar way to the repeated test strategy described in the previous section in that we are executing each test case several times looking for a failure, but instead of blindly rerunning the test hoping for a random schedule that reveals a race condition, we choose schedules that are likely to do so. Thus we would expect shrinking with procrastination to outperform repeated testing with random schedules. This is indeed the case as shown by the shrinking results for procrastination in Figure 6.5.

We can see that using procrastination we find minimal test cases around 60% of the time for all of the parameter values that we tried, but the time spent shrinking varies a lot. In this example, it seems that sticking to first order procrastination and only trying a small number of conflicting actions is most efficient. In fact this is faster than any of the previously attempted shrinking strategies, and reaching the same quality of shrinking using repeated tests would take somewhere around 2 to 3 times longer.

Which parameters to give when procrastinating depends very much on the nature of the race condition: If the race condition is present in many test cases but requires very specific scheduling to be revealed we need thorough procrastination. If, on the other hand, few test cases contain the race condition—which is to say that many of the candidate shrunk test cases will not be able to provoke the bug—then thorough procrastination will waste a lot of time on test cases that cannot fail. In our example only around 8% of the test cases contain the race condition and finding it when it is present seems to be relatively easy, so we can get away with less thorough procrastination.

It is interesting to note that higher order procrastination does im-

prove the quality of shrinking, in particular in the worst case, but it is not obviously better than simply repeating each test 100 times.

6.3 Reusing old schedules when shrinking

When implementing procrastination we had to modify PULSE to enable it to follow schedules that contain infeasible decisions. This led us to another idea for improving shrinking: what if, instead of running the smaller test cases on completely random schedules, we reused the schedule that made the bigger test case fail. That is, we try as far as possible to repeat the scheduling decisions that we know led to a failure for the bigger test case. The rationale for this is that if the smaller test case still contains the race condition, then the scheduling decisions that revealed the race condition for the bigger test case are likely to be valid also for the smaller one.

Naming of processes An issue that showed up when we started to reuse schedules during shrinking, that is not present for procrastination was that we had to be more careful about how processes are named. If a schedule says to deliver a message from process `A` to process `B` it is important that we use the names `A` and `B` for these processes when reusing that schedule.

The names are chosen automatically by PULSE. With our default naming scheme, the name chosen for a process depends on the names of the processes that have already been spawned, since each process needs a unique name. For instance, in the "world hello" example the first process to be spawned in the proxy chain is named `root-proxy`, the second `root-proxy1`, then `root-proxy2` and so on. This means that if a shrinking step removes the spawning of a process that could affect the names of processes that are spawned later in the test, and the schedule from the original test case would not make much sense for the shrunk test case, because the shrunk test case and the original test case would name their processes differently. To solve this problem we record in the schedule when a process is spawned and what name it's given. When following an existing schedule PULSE then reads the process name from the schedule instead of generating a fresh one. This is not a perfect solution, but it is a significant improvement over the previous situation.

Results In the process registry case reusing the schedule during shrinking turns out to work amazingly well (see Figure 6.6). Shrinking is more than 3 times faster than using procrastination and the shrunk test cases

Level	Limit	Time (s)	Avg. Len.	Max. Len.	Minimal
0	$\langle\rangle$	0.74	5.3	14	33%
1	$\langle\infty\rangle$	3.31	4.6	9	80%
1	$\langle 10\rangle$	1.87	4.6	9	78%
2	$\langle 5, 1\rangle$	5.93	4.6	9	86%
2	$\langle 10, 3\rangle$	10.08	4.6	10	84%
2	$\langle 10, 10\rangle$	12.03	4.6	10	82%
2	$\langle 20, 20\rangle$	21.74	4.6	9	83%

Figure 6.6: Shrinking performance with schedule reuse.

are as small as what we obtained using procrastination. The number of minimal test cases, however, is surprisingly low. The reason for this is that schedule reuse works poorly when shrinking moves commands from one process to another, for instance moving a command from one of the parallel sequences to the initial sequential part. This is because actions performed by the moved command will take place in a different process, so the corresponding scheduling decisions will be invalid. On the other hand it works very well for shrinking steps where the structure of the test case stays the same and we just remove unnecessary commands.

Note that in this case we did not repeat any tests during shrinking. When reusing the schedule this makes less sense, since each repeated test would start from the same schedule. It could still be of some benefit since PULSE reverts to random scheduling when there are no more feasible scheduling decisions.

Combining schedule reuse with procrastination improves both the quality and speed of shrinking, the proportion of minimal test cases goes from 60% to 80% compared to procrastination without schedule reuse and the worst case is significantly improved.

It is still more than twice as fast to just use schedule reuse when shrinking without compromising the quality of shrinking significantly. The reason why procrastination is so much slower is that it is spending quite a lot of time procrastinating shrunk test cases that cannot fail. One possible improvement would be to interleave the procrastinations for all shrinkings, that is, first try each smaller test case once, and if none of them fails try one procrastination for each case, and so on. This should result in similar performance to schedule reuse with no procrastination,

but with procrastination's quality. We leave the implementation of this strategy to future work.

7 Related Work

Much of the work regarding race condition detection has been focused around imperative (object oriented) programming languages, such as C, C++, and Java. In these languages a data race, in its simplest form, is when a shared piece of data is accessed and updated in a malign pattern. Several techniques have been proposed for detection of such data races, like [O'Callahan and Choi, 2003] and [Savage et al., 1997]. A problem, however, is that a large portion of the potential data races found are benign. As a result, research effort has focused on atomicity and/or serializability violations [Flanagan and Freund, 2004, Wang and Stoller, 2006]. By defining *units of work* to be atomic/serializable, it is possible to more accurately detect true concurrency bugs.

A problem with the previously mentioned techniques is that they do not take into account, potential correlations between the shared variables. Thus, it is possible to miss high-level data races and also to report false warnings. Vaziri et al. [Vaziri et al., 2006] address this problem with a more involved correctness criterion, namely *atomic-set serializability*. All of the race conditions above, simple data races, high-level data races, atomicity, and serializability, can be characterized by atomic-set serializability. The idea is to choose (parts of) the (object) state as atomic sets and methods/functions as units of work, and identify problematic interleavings of units of work in relation to the atomic sets.

Atomic-set serializability has proved to be accurate in discovering true concurrency bugs [Kidd et al., 2009], and the tool AssetFuzzer [Lai et al., 2010] is based on this technique.

Techniques where the scheduling is randomized have successfully been tried for concurrent, multi-threaded, programs [Stoller, 2002]. Although effective, the simple random technique is depending on the fact that harmful schedules are not too hard to find. To improve the situation, active randomized scheduling is used. The idea was introduced by RaceFuzzer [Sen, 2008], and further improved by AssetFuzzer [Lai et al., 2010]. The technique is similar to our procrastination: the program is run, the obtained schedule is analyzed for potential conflicts, and finally the program is re-run with a schedule that is (more) likely to trigger a conflict. A problem with this technique is *thrashing* [Joshi et al., 2009], where tests are failing because the calculated, poten-

tially problematic, schedule is not feasible and the program dead-locks. Luckily, with our relaxed schedules, this is not a problem for us.

In contrast to using a race condition detection criterion, and detecting violations in terms of crashes or memory access patterns, we use properties to decide whether a test has passed or not. When `parallel_commands` is used with QuickCheck, PULSE, and procrastination, the end result is close functionality-wise to AssetFuzzer.

Another often used approach is to systematically try all possible schedulings, either in the form of ordinary model checking or a repeated-test strategy. CHESS [Musuvathi et al., 2008] uses the latter, it systematically generates all (non-equivalent) interleavings for a given test scenario. By using some model checking techniques the number of explored interleavings can be kept at a reasonable level. For Erlang programs, McErlang is a traditional model checker that can find concurrency bugs [Fredlund and Svensson, 2007]. Although, in theory, the idea to explore all possible execution paths is tempting, there are usually problem for these methods to scale to larger programs. CHESS reports on some success [Musuvathi et al., 2008] on this issue, while Kidd et al. [Kidd et al., 2007] concludes that their approach is not scalable even for medium-sized programs.

8 Conclusions

Testing concurrent programs is difficult, even when the program is written in Erlang. We have shown how we can introduce procrastination into the earlier developed user-level scheduler PULSE. With procrastination, potential race conditions are detected in a given schedule and a new schedule is computed to provoke such potential races. We changed PULSE in such a way that it can run infeasible schedules, i.e., it runs a schedule as far as possible and continues with random scheduling when the given schedule cannot be followed any longer. This allowed us to use computationally inexpensive heuristics to compute new schedules from a schedule with a potential race condition. Nevertheless, our procrastination is very effective in provoking race conditions as our empirical data shows.

We use procrastination in combination with QuickCheck. Test cases are automatically generated and by executing PULSE with procrastination, we are able to find race conditions effectively. A beneficial side-effect of the changes made to PULSE, is that in order to facilitate the usage of infeasible schedules, we are also able to re-use a schedule

for a *different* test case. This can be exploited while shrinking, where we can now re-use the failing schedule when we test a shrunk, very similar test case. For the running example of this paper, this turned out to be amazingly effective. In fact, rather unexpectedly, just re-using of the schedule and no procrastination during shrinking gave the quickest shrinking, albeit not the smallest counterexamples. In our further research we will examine additional examples in order to develop a good strategy in mixing procrastination and schedule reuse during shrinking in order to quickly find the smallest counterexamples.

Acknowledgements

This research was sponsored by EU FP7 Collaborative project *ProTest*, grant number 215868.

Bibliography

A. Abel, T. Coquand, and U. Norell. Connecting a logical framework to a first-order logic prover. In Gramlich [2005], pages 285–301. ISBN 3-540-29051-6.
→ 1 citation on page: 91

J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.
→ 4 citations on 3 pages: 162, 164, and 197

C. Artho, K. Havelund, and S. Honiden. Visualization of concurrent program executions. In *COMPSAC '07: Proc. of the 31st Annual International Computer Software and Applications Conference*, pages 541–546, Washington, DC, USA, 2007. IEEE Computer Society.
→ 1 citation on page: 193

T. Arts and L.-Å. Fredlund. Trace analysis of Erlang programs. *SIGPLAN Notices*, 37(12):18–24, 2002.
→ 1 citation on page: 193

T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with Quviq QuickCheck. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 2–10, New York, NY, USA, 2006. ACM. ISBN 1-59593-490-1.
→ 7 citations on 7 pages: 6, 19, 66, 162, 165, 197, and 199

L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation. *Inf. Comput.*, 121(2):172–192, 1995.
→ 1 citation on page: 151

J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321136160.

→ 1 citation on page: 8

M. Barnett, B.-Y. E. Chang, R. DeLine, B. J. 0002, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005. ISBN 3-540-36749-7.
→ 1 citation on page: 4

J. L. Bentley. *Programming pearls*. Addison-Wesley, 1986. ISBN 978-0-201-10331-1.
→ 1 citation on page: 3

C. Benzmüller, L. C. Paulson, F. Theiss, and A. Fietzke. LEO-II: A cooperative automatic theorem prover for higher-order logic. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR 2008*, volume 5195 of *LNAI*, pages 162–170. Springer, 2008.
→ 1 citation on page: 158

R. S. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998. ISBN 0-13-484346-0.
→ 1 citation on page: 12

J. C. Blanchette and A. Krauss. Monotonicity inference for higher-order formulas. *J. Autom. Reasoning*, 47(4):369–398, 2011.
→ 3 citations on 3 pages: 16, 92, and 158

J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-14051-8.
→ 1 citation on page: 5

J. C. Blanchette and A. Paskevich. TFF1: The TPTP typed first-order form with rank-1 polymorphism. Tech. report, http://www21.in.tum.de/~blanchet/tff1spec.pdf, 2012.
→ 6 citations on 5 pages: 111, 112, 113, 153, and 158

J. C. Blanchette and A. Popescu. Formal development associated with Paper IV.

http://www21.in.tum.de/~popescua/enc_types_devel.zip, 2012.

→ 1 citation on page: 111

J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In N. Børner and V. Sofronie-Stokkermans, editors, *Automated Deduction*, volume 6803 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2011.
→ 3 citations on 3 pages: 18, 112, and 153

J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Empirical data associated with Paper IV. http://www21.in.tum.de/~blanchet/enc_types_data.tar.gz, 2012.
→ 1 citation on page: 153

F. Bobot and A. Paskevich. Expressing polymorphic types in a many-sorted language. In C. Tinelli and V. Sofronie-Stokkermans, editors, *FroCoS 2011*, volume 6989 of *LNAI*, pages 87–102. Springer, 2011.
→ 2 citations on 2 pages: 134 and 157

F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing polymorphism in SMT solvers. In C. Barrett and L. de Moura, editors, *SMT 2008*, 2008.
→ 1 citation on page: 158

F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In K. R. M. Leino and M. Moskal, editors, *Boogie 2011*, pages 53–64, 2011.
→ 3 citations on 3 pages: 4, 157, and 158

S. Böhme and T. Nipkow. Sledgehammer: Judgement Day. In J. Giesl and R. Hähnle, editors, *IJCAR 2010*, volume 6173 of *LNAI*, pages 107–121. Springer, 2010.
→ 2 citations on 2 pages: 151 and 153

M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 169–180, New York, NY, USA, 2006. ACM. ISBN 1-59593-263-1.
→ 1 citation on page: 65

C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. C. Rinard. Using first-order theorem provers in the Jahob data structure verification system. In B. Cook and A. Podelski, editors, *VMCAI 2007*, volume 4349 of *LNCS*, pages 74–88. Springer, 2007.
→ 2 citations on 2 pages: 125 and 158

A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: meta-level guidance for mathematical reasoning*. Cambridge University Press, New York, NY, USA, 2005.
→ 2 citations on 2 pages: 13 and 77

F. W. Burton. An efficient functional implementation of FIFO queues. *Inf. Process. Lett.*, 14(5):205–206, 1982.
→ 1 citation on page: 27

R. Carlsson and D. Gudmundsson. The new array module. In B. Däcker, editor, *13th International Erlang/OTP User Conference*, Stockholm, 2007. Available from `http://www.erlang.se/euc/07/`.
→ 1 citation on page: 29

H. R. Chamarthi, P. Dillinger, P. Manolios, and D. Vroon. The ACL2 sedan theorem proving system. In *TACAS*, pages 291–295, 2011.
→ 3 citations on 3 pages: 13, 80, and 84

K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6.
→ 10 citations on 10 pages: 4, 5, 24, 50, 66, 70, 86, 162, 165, and 199

K. Claessen and J. Hughes. Testing monadic code with QuickCheck. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 65–77, New York, NY, USA, 2002. ACM. ISBN 1-58113-605-6.
→ 1 citation on page: 66

K. Claessen and A. Lillieström. Automated inference of finite unsatisfiability. *Journal of Automated Reasoning*, 2011. ISSN 0168-7433. 10.1007/s10817-010-9216-8.
→ 3 citations on 3 pages: 107, 127, and 150

K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model finding. In *Proc. of Workshop on Model Computation (MODEL)*, 2003.
→ 1 citation on page: 105

K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. HipSpec source code repository. http://www.github.com/danr/hipspec, 2013a.
→ 1 citation on page: 80

K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. HipSpec evaluation results. http://www.cse.chalmers.se/~danr/hipspec, 2013b.
→ 3 citations on 2 pages: 80 and 82

J.-F. Couchot and S. Lescuyer. Handling polymorphism in automated deduction. In F. Pfenning, editor, *CADE-21*, volume 4603 of *LNAI*, pages 263–278. Springer, 2007.
→ 1 citation on page: 158

M. Cronqvist. Troubleshooting a large Erlang system. In *ERLANG '04: Proc. of the 2004 ACM SIGPLAN workshop on Erlang*, pages 11–15, New York, NY, USA, 2004. ACM.
→ 1 citation on page: 162

L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
→ 4 citations on 4 pages: 3, 11, 80, and 153

E. W. Dijkstra. The Humble Programmer (EWD 340). http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html, 1972. URL http://www.cs.utexas.edu/~{}EWD/transcriptions/EWD03xx/EWD340.html.
→ 1 citation on page: 2

L. Dixon and J. D. Fleuriot. Higher order rippling in IsaPlanner. In *Theorem Proving in Higher Order Logics*, volume 3223 of *LNCS*, pages 83–98, 2004.
→ 3 citations on 3 pages: 13, 70, and 84

H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, second edition, January 2001. ISBN 0122384520.
→ 3 citations on 3 pages: 90, 117, and 157

M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007. ISSN 0167-6423.
→ 1 citation on page: 64

N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
→ 1 citation on page: 103

C. Flanagan and S. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 269, April 2004.
→ 1 citation on page: 217

L.-Å. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. *SIGPLAN Not.*, 42 (9):125–136, 2007.
→ 3 citations on 3 pages: 6, 191, and 218

L.-Å. Fredlund and H. Svensson. Integration of QuickCheck and McErlang. https://babel.ls.fi.upm.es/trac/McErlang/wiki/QuickCheck/McErlang, 2010.
→ 1 citation on page: 6

E. R. Gansner and S. C. North. An open graph visualization system and its applications. *Software - Practice and Experience*, 30: 1203–1233, 1999.
→ 1 citation on page: 181

B. Gramlich, editor. *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*, volume 3717 of *Lecture Notes in Computer Science*, 2005. Springer. ISBN 3-540-29051-6.
→ 2 citations on 2 pages: 220 and 232

J. Henkel, C. Reichenbach, and A. Diwan. Discovering documentation for Java container classes. *IEEE Trans. Software Eng.*, 33(8): 526–543, 2007.
→ 3 citations on 3 pages: 10, 32, and 64

M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *POPL '87: Proc. of the 14th ACM SIGACT-SIGPLAN symposium on Principles of Prog. Lang.*, pages 13–26, New York, NY, USA, 1987. ACM.
→ 1 citation on page: 169

T. Hillenbrand, A. Buch, R. Vogt, and B. Löchner. Waldmeister - high-performance equational deduction. *J. Autom. Reason.*, 18(2): 265–270, Apr. 1997. ISSN 0168-7433.
→ 1 citation on page: 11

C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.
→ 1 citation on page: 54

J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.
→ 1 citation on page: 8

J. Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, pages 53–96. Springer Verlag, LNCS 925, 1995.
→ 1 citation on page: 9

J. Hughes. QuickCheck testing for fun and profit. In M. Hanus, editor, *PADL*, volume 4354 of *Lecture Notes in Computer Science*, pages 1–32. Springer, 2007. ISBN 978-3-540-69608-7.
→ 4 citations on 4 pages: 66, 162, 165, and 199

J. Hughes and H. Bolinder. Testing a database for race conditions with QuickCheck. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, Erlang '11, pages 72–77, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0859-5.
→ 1 citation on page: 6

J. Hughes, U. Norell, and J. Sautret. Using temporal relations to specify and test an instant messaging server. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 95–102, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-970-1.
→ 1 citation on page: 6

J. Hurd. First-order proof tactics in higher-order logic theorem provers. In M. Archer, B. Di Vito, and C. Muñoz, editors, *Design and*

Application of Strategies/Tactics in Higher Order Logics, number CP-2003-212448 in NASA Tech. Reports, pages 56–68, 2003.
→ 1 citation on page: 151

A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16:16–1, 1995.
→ 4 citations on 4 pages: 13, 71, 81, and 84

D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *In Proc. of the 19th International Conference on Software Engineering*, pages 360–370, 1997.
→ 1 citation on page: 193

M. Johansson, L. Dixon, and A. Bundy. Case-analysis for rippling and inductive proof. In *ITP*, pages 291–306, 2010a.
→ 1 citation on page: 80

M. Johansson, L. Dixon, and A. Bundy. Lemma discovery and middle-out reasoning for automated inductive proof. In S. Siegler and N. Wasser, editors, *Induction, Verification and Termination Analysis: Festschrift for Christoph Walther*, volume 6463 of *LNAI*, pages 102–116. Springer, 2010b.
→ 1 citation on page: 85

M. Johansson, L. Dixon, and A. Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 47(3):251–289, 2011.
→ 5 citations on 5 pages: 13, 71, 82, 83, and 85

P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 110–120, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1.
→ 1 citation on page: 217

M. Kaufmann, M. Panagiotis, and J. Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
→ 2 citations on 2 pages: 70 and 84

N. Kidd, T. Reps, J. Dolby, and A. Vaziri. Static detection of atomic-set-serializability violations. In *Technical Report 1623*, University of Wisconsin-Madison, 2007.

→ 1 citation on page: 218

N. Kidd, T. Reps, J. Dolby, and M. Vaziri. Finding concurrency-related bugs using random isolation. In N. Jones and M. Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *Lecture Notes in Computer Science*, pages 198–213. Springer Berlin / Heidelberg, 2009.
→ 1 citation on page: 217

G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an operating system kernel. *CACM*, 53(6):107–115, Jun 2010.
→ 1 citation on page: 3

P. N. Klein, R. H. B. Netzer, and H.-I. Lu. Detecting race conditions in parallel programs that use semaphores. *Algorithmica*, 35(4): 321–345, 2003.
→ 1 citation on page: 190

K. Korovin. Instantiation-based automated reasoning: From theory to practice. In R. A. Schmidt, editor, *CADE-22*, volume 5663 of *LNAI*, pages 163–166. Springer, 2009.
→ 1 citation on page: 153

Z. Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 235–244, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6.
→ 4 citations on 3 pages: 202, 204, and 217

L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
→ 1 citation on page: 185

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.
→ 2 citations on 2 pages: 19 and 169

C. Le Goues, K. R. M. Leino, and M. Moskal. The Boogie verification debugger. In *Proceedings of the 9th international conference on*

Software Engineering and Formal Methods, SEFM'11, pages 407–414, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24689-0.
→ 1 citation on page: 5

K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-17510-4, 978-3-642-17510-7. URL http://dl.acm.org/citation.cfm?id=1939141.1939161.
→ 3 citations on 3 pages: 8, 80, and 85

K. R. M. Leino and G. Nelson. An extended static checker for modula-3. In K. Koskimies, editor, *Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-64304-3. doi: 10.1007/BFb0026441. URL http://dx.doi.org/10.1007/BFb0026441.
→ 1 citation on page: 5

K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In J. Esparza and R. Majumdar, editors, *TACAS 2010*, volume 6015 of *LNCS*, pages 312–327. Springer, 2010.
→ 4 citations on 2 pages: 157 and 158

X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. URL http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf.
→ 1 citation on page: 3

S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGARCH Comput. Archit. News*, 36(1):329–339, 2008.
→ 2 citations on 2 pages: 170 and 190

S. Maoz, A. Kleinbort, and D. Harel. Towards trace visualization and exploration for reactive systems. In *VLHCC '07: Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 153–156, Washington, DC, USA, 2007. IEEE Computer Society.
→ 1 citation on page: 193

B. Marick. How to misuse code coverage. http://www.exampler.com/testing-com/writings/coverage.pdf, 1999.
→ 1 citation on page: 2

The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL http://coq.inria.fr. Version 8.0.
→ 1 citation on page: 3

R. L. McCasland and A. Bundy. MATHsAiD: a mathematical theorem discovery tool. In *Proceedings of the Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'06)*, 2006.
→ 2 citations on 2 pages: 65 and 71

S. McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004. ISBN 0735619670, 9780735619678.
→ 2 citations on page: 1

J. Meng and L. C. Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60, 2008.
→ 8 citations on 8 pages: 91, 109, 112, 117, 118, 149, 150, and 157

O. Montano-Rivas, R. McCasland, L. Dixon, and A. Bundy. Scheme-based theorem discovery and concept invention. *Expert Systems with Applications*, 39(2):1637–1646, 2012.
→ 6 citations on 6 pages: 13, 71, 82, 83, 84, and 85

S. Muggleton and L. de Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19:629–679, 1994.
→ 1 citation on page: 65

M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association. URL http://portal.acm.org/citation.cfm?id=1855741.1855760.
→ 4 citations on 3 pages: 20, 192, and 218

G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. ISBN 0471469122.
→ 1 citation on page: 1

R. H. B. Netzer and B. P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *In Proc. of the 1990 Int. Conf. on Parallel Processing*, pages 93–97, 1990.
→ 1 citation on page: 190

R. Nieuwenhuis and A. Oliveras. Proof-producing congruence closure. In *RTA '05*, pages 453–468, Nara, Japan, 2005. Springer LNCS.
→ 1 citation on page: 38

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
→ 3 citations on 3 pages: 3, 84, and 111

R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. ACM. ISBN 1-58113-588-2.
→ 1 citation on page: 217

A. J. Offutt. A practical system for mutation testing: help for the common programmer. In *Proceedings of the 1994 international conference on Test*, ITC'94, pages 824–830, Washington, DC, USA, 1994. IEEE Computer Society. ISBN 0-7803-2102-2. URL http://dl.acm.org/citation.cfm?id=1895949.1896084.
→ 1 citation on page: 2

C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998. ISBN 0-521-66350-4.
→ 1 citation on page: 51

C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *SIGSOFT '08/FSE-16: Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 135–145, New York, NY, USA, 2008. ACM.
→ 1 citation on page: 191

L. C. Paulson. A generic tableau prover and its integration with Isabelle. *Journal of Universal Computer Science*, 5(3):73–87, 1999.

→ 1 citation on page: 91

L. C. Paulson and K. W. Susanto. Source-level proof reconstruction for interactive theorem proving. In K. Schneider and J. Brandt, editors, *TPHOLs 2007*, volume 4732 of *LNCS*, pages 232–245. Springer, 2007.
→ 1 citation on page: 151

S. Ranise, C. Ringeissen, and C. G. Zarba. Combining data structures with nonstably infinite theories using many-sorted logic. In Gramlich [2005], pages 48–64. ISBN 3-540-29051-6.
→ 1 citation on page: 92

J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM.
→ 1 citation on page: 14

A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Comm.*, 15(2-3):91–110, 2002.
→ 2 citations on 2 pages: 11 and 153

D. Rosén. Proving Equational Haskell Properties using Automated Theorem Provers, 2012. MSc. Thesis, University of Gothenburg.
→ 3 citations on 3 pages: 11, 71, and 73

S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15:391–411, November 1997. ISSN 0734-2071.
→ 2 citations on 2 pages: 202 and 217

S. Schulz. System description: E 0.81. In D. Basin and M. Rusinowitch, editors, *IJCAR 2004*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.
→ 2 citations on 2 pages: 11 and 153

K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 11–21, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2.
→ 5 citations on 5 pages: 20, 21, 192, 198, and 217

W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: A tool for the automatic verification of algebraic properties of functional programs. Technical report, Imperial College London, February 2011. URL http://pubs.doc.ic.ac.uk/zeno/.
→ 4 citations on 4 pages: 13, 70, 80, and 84

M. E. Stickel. Schubert's steamroller problem: Formulations and solutions. *J. Autom. Reasoning*, 2(1):89–101, 1986.
→ 3 citations on 3 pages: 110, 117, and 157

S. D. Stoller. Testing concurrent Java programs using randomized scheduling. *Electronic Notes in Theoretical Computer Science*, 70(4): 142 – 157, 2002. ISSN 1571-0661. URL http://www.sciencedirect.com/science/article/B75H1-4DDWJNG-9H/2/00f38147773160fadc7702f42759ab59. RV'02, Runtime Verification 2002 (FLoC Satellite Event).
→ 1 citation on page: 217

G. Sutcliffe. The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.
→ 5 citations on 5 pages: 89, 106, 116, 127, and 153

G. Sutcliffe. Proceedings of the 6th IJCAR ATP system competition (CASC-J6). In G. Sutcliffe, editor, *CASC-J6*, volume 11 of *EPiC*, pages 1–50. EasyChair, 2012.
→ 1 citation on page: 157

H. Svensson and L.-Å. Fredlund. A more accurate semantics for distributed Erlang. In *Erlang '07: Proc. of the 2007 SIGPLAN Erlang Workshop*, pages 43–54, New York, NY, USA, 2007. ACM.
→ 1 citation on page: 176

G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002.
→ 1 citation on page: 1

B. Topol, J. Stasko, and V. Sunderam. Integrating visualization support into distributed computing systems. *Proc. of the 15th Int. Conf. on: Distributed Computing Systems*, pages 19–26, May-Jun 1995.
→ 1 citation on page: 192

J. Urban. MPTP 0.2: Design, implementation, and initial experiments. *J. Autom. Reasoning*, 37(1-2):21–43, 2006.
→ 1 citation on page: 157

M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 334–345, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2.
→ 2 citations on 2 pages: 197 and 217

D. Vytiniotis, D. Rosén, S. P. Jones, and K. Claessen. HALO: Haskell to logic through denotational semantics. In *Proceedings of POPL'13*. ACM, 2013.
→ 1 citation on page: 86

L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering*, 32:93–110, 2006. ISSN 0098-5589.
→ 1 citation on page: 217

C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1965–2013. Elsevier, 2001.
→ 1 citation on page: 153

C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobalt, and D. Topi. SPASS version 2.0. In A. Voronkov, editor, *Automated deduction, CADE-18 : 18th International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 275–279, Copenhagen, Denmark, 2002. Springer. ISBN 3-540-43931-5.
→ 1 citation on page: 90

M. Wenzel. Type classes and overloading in higher-order logic. In E. L. Gunter and A. Felty, editors, *TPHOLs 1997*, volume 1275 of *LNCS*, pages 307–322. Springer, 1997.
→ 1 citation on page: 149

C. A. Wick and W. McCune. Automated reasoning about elementary point-set topology. *Journal of Automated Reasoning*, 5(2):239–255, 1989.
→ 6 citations on 6 pages: 90, 91, 117, 123, 130, and 157

U. T. Wiger. Extended process registry for Erlang. In *ERLANG '07: Proc. of the 2007 SIGPLAN workshop on ERLANG Workshop*, pages 1–10, New York, NY, USA, 2007. ACM.
→ 2 citations on 2 pages: 163 and 205

A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, Oct. 2005. ISBN 1558608664.
→ 1 citation on page: 2