RANDOM TESTING FOR LANGUAGE DESIGN

Leonidas Lampropoulos

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2018

Supervisor of Dissertation

Benjamin C. Pierce
Professor, Computer and Information Science

Graduate Group Chairperson

Lyle Ungar, Professor, Computer and Information Science

Dissertation Committee:

Steve Zdancewic, Professor of Computer and Information Science

Stephanie Weirich, Professor of Computer and Information Science

Majur Naik, Professor of Computer and Information Science

John Hughes, Professor of Computing Science, Chalmers University of Technology, Gothenburg

RANDOM TESTING FOR LANGUAGE DESIGN

Acknowledgments

I want to begin by thanking my advisor, Benjamin Pierce, who was as great an advisor as anyone could hope to have. I've learned so much by working with you and alongside you that I would be happy to eventually pass on even a fraction to my own students.

I also want to extend my thanks to Stephanie Weirich and Steve Zdancewic for hosting everyone in their home so many times and showing me that one can remain fun at the highest level. To John Hughes for being a great source of inspiration throughout my testing adventures. To Majur Naik for joining my committee and his flexibility that made actually picking a defense date possible. To Dimitrios Vytiniotis and Aditya Nori for giving me a chance to work with them on something completely different during my summer internship. To Catalin Hriţcu for being a wonderful collaborator.

I have to express extreme gratitude towards my undergrad advisor, Kostis Sagonas, for introducing me to the magic world of programming languages, guiding me towards my path at Penn, and being a great source of advice whenever I needed it for almost 10 years. And to Nikos Papaspyrou, who gave me the bug for programming contests and set an example of the kind of teacher and person I aspire to be.

Throughout my years at Penn, I was fortunate to interact with many more wonderful people. Thanks to Antal Spector-Zabusky, who has made every one of my papers prettier with his LaTeX wizardry. To Arthur Azevedo de Amorim, who made all my proof attempts easier by introducing me to ssreflect. To Kenny Foner for his punny paper titles, and also to Hengchu Zhang for including me in their crazy StrictCheck idea. To Jennifer Paykin and Robert Rand, who have been

helping since our first year together. And to the entirety of the Penn PL Club for their amazing feedback in my talks and projects over the years. And of course, to Nikos Vasilakis, who has been as amazing a friend as he has been a coworker.

My PhD would not have been possible without my friends in the Greek PL chat. Thanks to Zoe Paraskevopoulou, who was fundamental in the work presented in this thesis. To Niki Vazou and her amazing academic matchmaking skills. To George Karachalias and our all-encompassing chats. To Aggelos Biboudis for trying (and failing) to convince me that Haskell is not the only answer. And to Nick Giannarakis for sharing the same sports fandom.

My work all this years was only possible due to the love and support of my friends. I want say thanks to my friends in Philly: Mike, Stathis, Venetia, Pavlos, Melissa, Markos, Aris, Nasia, Aphrodite, Michael, Vlassis, Dafni, Alexia, Alex, Yannis, George, Tassos, Spiros, Kostas, Vasilis, Fragiskos, Giorgos and Ioanna, who made moving across the Atlantic fun. To the asado and cumbia crew: Santi, Luiz, Miguel, Fernando and Luana, who have elevated barbeques to an art form. To my friends from Penn Gamers: Chris (both the suspicious, agreeable and the evasive variety), Derek and Sid, who shared my passion of heavy board games. To my friends from undergrad: Despoina, Nikolas, Alexandros, Mary, Sergios, Ignatios, Nikos, Vasilis, Evridiki and Lydia, who embarked on a similar journey across the globe at the same time. And, of course, to my closest group of friends back home. Rigo, Ari, Christo, Andrea, Toli, Tasso, Vaggeli, Kate and Anneza: you make my life great just by being in it.

Last, but certainly not least, I want to thank my parents, Sofia and Konstantinos, and my amazing sister, Elena, who have always been there through good and bad. I love you and your support means everything to me.

ABSTRACT

RANDOM TESTING FOR LANGUAGE DESIGN
Leonidas Lampropoulos
Benjamin C. Pierce

Property-based random testing can facilitate formal verification, exposing errors early on in the proving process and guiding users towards correct specifications and implementations. However, effective random testing often requires users to write custom generators for well-distributed random data satisfying complex logical predicates, a task which can be tedious and error prone.

In this work, I aim to reduce the cost of property-based testing by making such generators easier to write, read and maintain. I present a domain-specific language, called *Luck*, in which generators are conveniently expressed by decorating predicates with lightweight annotations to control both the distribution of generated values and the amount of constraint solving that happens before each variable is instantiated.

I also aim to increase the applicability of testing to formal verification by bringing advanced random testing techniques to the Coq proof assistant. I describe QuickChick, a QuickCheck clone for Coq, and improve it by incorporating ideas explored in the context of Luck to automatically derive provably correct generators for data constrained by inductive relations.

Finally, I evaluate both QuickChick and Luck in a variety of complex case studies from programming languages literature, such as information-flow abstract machines and type systems for lambda calculi.

Contents

List of Illustrations

Chapter 1

Introduction

Software errors are becoming an increasingly important problem as our society grows more and more reliant on computer systems. With formal verification, we can guarantee the absence of such errors in software artifacts by *proving* that these artifacts adhere to a formal specification. Unfortunately, even though recent advances in verification technology have allowed for incredible feats of proof engineering, like CompCert [85] (a verified optimizing compiler for C) and CertiKOS [57] (an extensible architecture for building certified OS kernels), such endeavors are still very expensive. It is not uncommon to spend weeks attempting to prove a theorem in a proof assistant only to discover that it is not actually true, wasting valuable time and effort.

Random testing techniques provide an interesting alternative. On the one hand, we can use such techniques to find bugs in extremely complex software. For instance, the CSmith project [127] uncovered more than 400 bugs in various C compilers, including GCC and LLVM. On the other hand, we cannot use them to guarantee the absence of such bugs. Nevertheless, we can still leverage random testing to aid our verification efforts, by revealing bugs early on in the process and helping in the discovery of complex program invariants.

A particularly effective form of random testing is property-based random testing (PBT). Popularized by QuickCheck [33], PBT is a semi-automatic technique that has since been adopted in a variety of programming languages [2, 87, 67, 100]

and theorem provers [27, 17, 99, 39]. At a high level, PBT tools require as input *properties* in the form of executable specifications, and generate random inputs in hopes of falsifying these properties. For example, consider the following specification and implementation of a `delete` function, written in Haskell: after deleting some number `x` from a list `l`, the result should not contain `x`. However, the implementation is faulty!

```
propDeleteCorrect :: Int -> [Int] -> Bool
propDeleteCorrect x l = not (member x (delete x l))

delete :: Int -> [Int] -> [Int]
delete _ [] = []
delete x (h:t) | x == h = t
               | otherwise = h : delete x t
```

To test this property, QuickCheck first generates an integer `x` and a list `l` randomly; it then executes `propDeleteCorrect`, and tests if the result is `True` or not. If it is, QuickCheck tries again until it reaches a prespecified number of successful tests; if not, QuickCheck reports the counterexample to the user.

In this case QuickCheck quickly discovers an error, reporting a bug when `x` is `0` and `l` is the two element list `[0,0]`:

```
*** Failed! Falsifiable (after 29 tests and 8 shrinks):
0
[0,0]
```

Indeed, the error in the recursive `delete` implementation is that when the head of the list is equal to the element to be deleted, we do not recursively `delete x` from its tail; this can only manifest if the list contains (at least) two copies of the element under deletion.

The counterexample reported above was minimal because of *shrinking*, a surprisingly underappreciated but crucial component of PBT: when a pair of falsifying inputs `(x,l)` is first discovered, it likely contains a lot of random noise that is irrelevant to the actual error. QuickCheck tries progressively smaller counterexamples

(for example, by dropping elements from the list). To put this in perspective, testing the exact same property and disabling shrinking altogether yields a practically unreadable counterexample:

```
*** Failed! Falsifiable (after 36 tests):
-57
[-47,30,-2,-57,-43,19,-45,-69,-15,-56,33,-57,-16,-44,55,-5,61]
```

Naturally, `-57` appears twice in the output list—can you spot where?

In general, random testing can be very effective out of the box, just like in this `delete` example. In such cases, it is far more time efficient to test and debug a property, gaining a certain degree of confidence in its correctness, and then attempt to prove it, than embarking on a costly journey of trying to prove `False`.

However, there is a large, important class of properties where random testing is less effective: properties involving preconditions. Consider for instance the following property of an `insert` function for ordered lists, which asserts that inserting an element into a sorted list yields a sorted result:

```
propInsertSorted x l = isSorted l ==> isSorted (insert x l)
```

In QuickCheck, `==>` is used to encode preconditions. To test such a property, QuickCheck generates random values for `x` and `l` and then *tests* if `isSorted l` holds. If the generated list is sorted, it proceeds to test the actual property (that the result of inserting an element into `l` is also sorted); if not, the generated values are discarded and the whole process starts from scratch. Especially for *sparse* predicates (i.e., ones that are rarely satisfied relative to their input type), this *generate-and-test* approach can be extremely inefficient. Even worse, it often provides unsatisfactory coverage, since only small inputs will satisfy the predicate. Indeed, testing `propInsertSorted` using QuickCheck's default behavior results in `l` being the empty list 40% of the time and a singleton list another 30%; clearly, such testing does not inspire confidence.

When dealing with preconditions, standard QuickCheck practice dictates writing *custom generators*: programs that produce a distribution of terms satisfying a

predicate (like `isSorted`) directly. QuickCheck provides a useful library of *generator combinators* to facilitate writing such programs and controlling the resulting distribution of terms—a crucial feature in practice [33, 65, 55]. The following code snippet shows a simple generator for sorted lists alongside the definition of `isSorted`:

```
isSorted :: [Int] -> Bool
isSorted [] = True
isSorted (x:l) = aux x l
  where aux x [] = True
        aux x (y:l) = x < y && aux y l

genSorted :: Gen [Int]
genSorted = oneof [ return []
                  , do x <- arbitrary
                       l <- auxG x
                       return (x:l) ]
  where auxG x = oneof [ return []
                       , do y <- genIntGreaterThan x
                            l <- auxG y
                            return (y : l) ]
```

The `isSorted` predicate has a standard recursive implementation in Haskell: an empty list is always sorted; a nonempty list is sorted if the head `x` is smaller than all the elements in the tail `l` and `l` itself is sorted, as computed by the auxiliary predicate `aux`.

The generator `genSorted` is written in QuickCheck's `Gen` monad, used to hide low-level plumbing of a random number generator. At a high level, `genSorted` closely follows the structure of `isSorted`. Just like `isSorted` first decides whether or not the list is empty by pattern matching, `genSorted` first chooses whether to *produce* an empty list (`return []`) or not. This is done using QuickCheck's `oneof` combinator (with type `[Gen a] -> Gen a`) that takes a list of generators and picks one at random to evaluate.

Similarly, like the nonempty branch of `isSorted` uses pattern matching to extract the head of the input list, the nonempty list generator first creates the head element `x` using `arbitrary` (a QuickCheck function that can be used to create random integers). It then uses an auxiliary generator `auxG`, which in turn is dual to `aux`: where `aux x` holds for lists that are both sorted and whose elements are greater than `x`, `auxG` *creates* such lists. The top-level pattern match in `aux` becomes another call to `oneof`, choosing between an empty list or a cons cell whose head `y` is greater than `x` and whose tail `l` is generated recursively to satisfy `aux y`.

Unfortunately, as predicates become more complex, coming up with efficient generators becomes increasingly more challenging—to the point of being a research contribution in its own right! For example, papers have recently been written about random generation techniques for well-typed lambda terms [103, 111, 43, 120] and for indistinguishable states of information-flow-control machines [65, 66]. Moreover, testing *invariant* properties (like well typedness for preservation of a type system) poses additional difficulties. Specifically, it requires both a generator, for obtaining inputs satisfying some predicate, and a *checker*, an executable form of the same predicate. These artifacts must then be kept in sync; the result is a maintenance nightmare that serves as a rich source of potential bugs.

If random testing is to be used to facilitate verification, discovering bugs through testing should be (much) faster than uncovering them by just attempting a proof directly. Whenever possible, testing feedback should be immediate with minimal user effort. The main goal of this work is to amplify the applicability of random testing in programming language design and verification.

As a first step, we implement the first fully-functional property-based testing tool for Coq, called QuickChick [36], initially as a complete clone of Haskell's QuickCheck. Using extraction to OCaml to take care of randomness and IO, QuickChick allows Coq users to enjoy the benefits of property-based testing, as users of other proof assistants have for quite some time.

We then expand on the natural idea of deriving generators automatically from predicates. Of course, this idea is not new; researchers have tried to generate testing inputs automatically using the structure of a predicate by borrowing ideas from both functional logic programming [44, 30, 110, 48, 31, 43] and constraint

solving [24, 116, 53, 79, 49, 117, 20, 5, 124]. However, in our experience, fully automatic methods soon reach a limit. Hand-tuned generators can be more efficient by an order of magnitude by exploiting domain-specific knowledge and, equally critically, by applying fine control over the distribution of generated inputs.

To further explore the design space of deriving generators while giving more control to the user, we develop a new domain-specific language, called *Luck*, that synergistically combines the complementary strengths and weaknesses of different approaches: in Luck, generators are written as lightly annotated predicates, with the annotations controlling both the distribution of generated values and the amount of constraint solving that happens before each variable is instantiated.

Finally, we improve upon QuickChick, adopting for it the ideas explored in the context of Luck. Specifically, we introduce an algorithm for deriving generators for data satisfying predicates in the form of dependently-typed inductive relations. Compared to Luck and similar techniques where the generation of data is performed by an interpreter, we *compile* inductive relations into generators in the host language. This in turn has multiple benefits. First, the generators themselves are compiled and optimized by the mature OCaml compiler, leading to increased performance. Second, we can provide certificates of correctness for each derived generator: that it is sound and complete with respect to the relation it was derived from. Last, but certainly not least, it gives users far more customizability. While it is possible to enjoy the push-button style automation by using the derived generators everywhere, it is also easy to mix and match, allowing experienced users to infuse domain specific knowledge for particularly tricky predicates that are beyond the scope of fully-automatic techniques.

1.1 Contributions

The rest of this thesis offers the following contributions:

- In Chapter 2, we introduce QuickChick [36], the first complete property-based random testing tool for the Coq proof assistant. QuickChick provides the full range of functionality of Haskell's QuickCheck, and additionally takes advantage of the dependently typed setting to offer possibilistic correctness guarantees for generator combinators, as discussed in our ITP 2015 paper [105]. This chapter also serves as a tutorial for QuickChick providing an in-depth explanation of its features and notations.

- Chapter 3 describes a case study in using random property-based testing to debug and inform the design of noninterfering information-flow abstract machines, a line of work that was presented in ICFP 2012 [65] and extended to a journal version [66]. In this case study, we explore different techniques for coming up with and implementing custom generators, identifying at the same time the key drawbacks of this approach: writing such generators for complex predicates can be both difficult and error-prone.

- In Chapter 4, we present Luck, a language for writing generators in the form of lightly annotated predicates, combining ideas from functional logic programming and constraint solving. The design of Luck is supported by a strong formal foundation; we evaluate this design by providing a prototype implementation. This work was the basis of the POPL 2017 paper "Beginner's Luck" [81].

- In Chapter 5, we enhance QuickChick with an automatic derivation procedure for random generators satisfying predicates in the form of inductive relations. We generalize the notion of narrowing from functional logic programming to obtain a compilation process from inductive definitions to both random generators satisfying them, and a proof of the generator's correctness—soundness and completeness—with respect to these definitions. We evaluate their efficiency in the same case studies as we evaluated Luck. This work was pre-

sented in the POPL 2018 paper "Generating Good Generators for Inductive Relations" [82].

- In Chapter 6, we address implementation concerns raised throughout the previous chapters. First, in Section 6.1, we describe a generic library for Coq, that was used to facilitate the derivation algorithm as a Coq plugin. Second, in Section 6.2, we develop a simple data structure, the *urn*, that allows sampling from updatable discrete distributions with asymptotically better performance than the currently available list-based solutions. The urn data structure was presented in Haskell Symposium 2018 [83].

- We explore related work in Chapter 7, while Chapter 8 concludes and draws several interesting directions for future work.

A lot of the work in this thesis is the product of collaboration with many wonderful collaborators. The end of each chapter includes a description of my specific contributions, as well as a list of the places the work involved has been presented.

Chapter 2

QuickChick

The first step towards integrating random testing and theorem proving in Coq is to create a proper testing tool for it. Prior efforts to bring such methods in Coq [126] were relatively simple and focused exclusively in aiding proof automation. In our approach, we aim for the full range of QuickCheck's functionality. Moreover, compared to existing random testing solutions for other proof assistants, we advocate a slightly different approach. In particular, Isabelle's QuickCheck [17], and, for more general language design frameworks, PLT Redex [43], provide a seamless push-button automation approach to testing. On the other hand, we wanted to allow expert users to be able to fully customize their generators, as hand-written fine-tuned generators are still an order of magnitude more efficient than state-of-the-art automated methods [31, 43]. At the same time, we want to provide similar automation support, by deriving such custom generators automatically, that users can inspect, read, use or modify at will.

In this chapter we introduce QuickChick, our own property-based testing tool for Coq. This chapter also serves as a tutorial for random testing using QuickChick, exploring all levels of its functionality. As a clone of Haskell's QuickCheck, a lot of the typeclass-based infrastructure is shared with the original. We describe both this infrastructure and the generalizations that were necessary to deal with Coq's `Prop` and unique typeclass implementation. However, before we describe this infrastructure, we need to address a rather unique issue of porting a random

testing tool in Coq: the need for *extraction*.

Haskell's QuickCheck is just a regular Haskell program that takes as input properties in the form of plain Haskell functions and tests them. It reports the current status at regular intervals before finally showing any counterexamples found to the user, all while requiring no support from the compiler. Unlike Haskell and most regular programming languages however, Coq does not support side effects. QuickChick, therefore, if written entirely in Gallina, would not be able to perform I/O and show testing feedback to users! Furthermore, despite a lot of recent progress, executing a program within Coq is slower than executing its counterpart in a traditional functional language. This efficiency gap is widened even further by the use of optimizing compilers.

To address both of these concerns, we turn to *extraction*. Extraction is a mechanism that transforms potentially dependently typed Coq proofs and programs into functional programs in another language. While various target languages are supported, including Haskell and Scheme, QuickChick extracts to OCaml: first, the extraction infrastructure for OCaml is the most mature and robust, and second, OCaml is already required for Coq and wouldn't pose an additional dependency for users.

2.1 Overview

In the introduction, we saw how one could test a simple `delete` function using Haskell's QuickCheck. In Gallina, the same faulty function can be similarly written as follows, where `beq_nat` is the library function that tests for standard natural number equality:

```
Fixpoint delete (x : nat) (l : list nat) : list nat :=
  match l with
    | [] => []
    | h::t => if beq_nat h x then t else h :: delete x t
  end.
```

The specification, however, can be expressed in two ways. Like before, the property can be expressed in executable form:

```
Lemma delete_correct_bool (x : nat) (l : list nat) :
  ~~ (List.member x (delete x l)).
```

To test such a property, QuickChick requires the same components as Haskell's QuickCheck:

1. *generators* for `x` and `l` to generate inputs,
2. *printers* for `x` and `l` to show counterexamples,
3. and *shrinkers* for `x` and `l` to minimize them.

However, specifications in Coq often appear in a propositional form, where instead of a predicate (`member : nat -> list nat -> bool`) an inductive relation (`In : nat -> list nat -> Prop`) is used:

```
Lemma delete_correct_prop : forall x l, ~ (In x (remove x l)).
```

To test this version of the property, QuickChick requires an additional component: a *checker*, i.e. a decidability procedure for `delete_correct_prop` to decide whether a generated input satisfies the desired property.

In the rest of this chapter we will look more closely at all different components and how QuickChick supports each one.

2.2 Generators

We will first take a look at random generators. At a high level, generators for some type `A` are just probability distributions over `A`. To enable efficient sampling from such distributions, we can represent them as functions from a *random seed* to `A`. To facilitate combining generators together in a compositional manner, we wrap this functional representation in a generator monad and provide a library of *generator combinators*.

2.2.1 Randomness

Unfortunately, Coq does not provide a library for creating, manipulating and consuming random seeds. Once again, we turn to extraction; QuickChick axiomatizes the type of random seeds.

```
Axiom RandomSeed : Type.
```

This command introduces an axiom, `RandomSeed`. It is then realized by extraction to the `Random.State.t` type of OCaml:

```
Extract Constant RandomSeed => "Random.State.t".
```

Thus, we can leverage the existing random infrastructure of the `Random.State` module of OCaml.

QuickChick provides two ways of obtaining random seeds; one relying on the self-initialization function of OCaml to obtain entropy in a system-dependent manner, and one where the user provides an initial integer seed, to allow for repeatable random draws.

```
Axiom newRandomSeed : RandomSeed.
Axiom mkRandomSeed  : Z -> RandomSeed.

Extract Constant newRandomSeed =>
  "(Random.State.make_self_init ())".
Extract Constant mkRandomSeed  =>
  "(fun x -> Random.init x; Random.get_state())".
```

In particular, QuickChick introduces a constant `newRandomSeed` which is extracted to a call to the self-initialization function `Random.State.make_self_init : unit -> Random.State.t`, which returns a different result every time. This obviously can be used to break referential transparency upon extraction, so it needs to be used with care.

Finally, we also need to use random seeds to produce useful data. The basis will be a function that receives a pair of natural numbers `x` and `y`, as well as a `RandomSeed`, and produces a random natural number in the range $[\mathtt{x}, \mathtt{y}]$, raising an error if $\mathtt{x} > \mathtt{y}$. In addition, it returns the modified random seed.

```
Axiom randomRNat  : nat  * nat -> RandomSeed -> nat * RandomSeed.
```

We can leverage OCaml's method for creating random integers, `Random.State.int`, which takes a seed `r` and a positive integer `n` and produces an integer between 0 (inclusive) and `n` (exclusive).

```
Extract Constant randomRNat  =>
  "(fun (x,y) r ->
      if y < x then failwith "..."
      else (x + (Random.State.int r (y - x + 1)), r))".
```

As `Random.State.int` modifies the input random seed in place, we just return the input seed `r`, giving the QuickChick user an illusion of a functional implementation.

2.2.2 Generator Combinators

The G Monad Building generators as functions from `RandomSeed` to a pair `A * RandomSeed` is possible, but would soon become extremely tedious. QuickChick wraps random seeds inside a reader monad, named `G`. Using the two basic functions `returnGen` and `bindGen`,

```
returnGen : forall {A}, A -> G A
bindGen   : forall {A B}, G A -> (A -> G B) -> G B,
```

where `A` and `B` denote universally quantified type parameters, we can provide a typeclass instance of the `Monad` typeclass: [1]

```
Instance gMonad : Monad G :=
  {
    ret  := @returnGen;
    bind := @bindGen
  }.
```

[1] This piece of code defines a constant `gMonad` with type `Monad G`. A record-like syntax is used to provide bindings for each typeclass method, as the implementation of typeclasses in Coq is based on records.

Primitive Generators Next, QuickChick provides generators for most primitive types, using `choose`:

```
choose : forall {A}, ChoosableFromInterval A -> A * A -> G A
```

The `choose` combinator uses a `ChoosableFromInterval` typeclass that describes primitive types `A`, for which it makes sense to randomly generate elements from a given interval, like natural numbers.

```
Class ChoosableFromInterval (A : Type) : Type :=
  {
    super : Ord A;
    randomR : A * A -> RandomSeed -> A * RandomSeed;
    ...
  }.
```

We have already seen a suitable implementation of the `randomR` method in the form of the `randomRNat` axiom realized via extraction. Similar instances are provided for other ordered primitive types, like integers and booleans.

Lists Since lists are arguably the most commonly used datatype in functional programming, QuickChick provides two list-specific combinators: `vectorOf` and `listOf`. The `vectorOf` combinator takes as input a natural number `n`, the length of the list to be generated, as well as a generator for elements of some type `A` and produces lists of `A`s. For example, if we sample the generator `vectorOf 3 (choose (0,4))` using the `Sample` command provided by QuickChick, we will only get lists of length 3 with elements between 0 and 4:

```
Sample (listOf (choose (0,4))).
```

```
                                                    *output*
[ [0, 1, 4], [1, 1, 0], [3, 3, 3], [0, 2, 1] ,
  [3, 3, 0], [3, 0, 4], [2, 3, 3], [3, 2, 4] ]
```

The second combinator, `listOf`, also requires a generator for elements of `A`, but no size argument:

```
Sample (listOf (choose (0,4))).
```

```
                                                                *output*
[ [ 0, 3, 2, 0 ],
  [ 1, 3, 4, 1, 0, 3, 0, 2, 2, 3, 2, 2, 2, 0, 4, 2, 3, 0, 1 ],
  [ 3, 4, 3, 1, 2, 4, 4, 1, 0, 3, 4, 3, 2, 2, 4, 4, 1 ],
  [ 0 ],
  [ 4, 2, 3 ],
  [ 3, 3, 4, 0, 1, 4, 3, 2, 4, 1 ],
  [ 0, 4 ],
  [  ],
  [ 1, 0, 1, 3, 1 ],
  [ 0, 0 ],
  ... ]
```

Which begs the question, how does `listOf` decide the size of the generated list?

The answer lies in the `G` monad. In addition to handling random-seed plumbing, the G monad also maintains a "current maximum size" : a natural number that can be used as an upper bound on the depth of generated objects. That is, internally, G A is just a synonym for `nat -> RandomSeed -> A`:

```
Inductive G (A:Type) : Type :=
| MkG : (nat -> RandomSeed -> A) -> G A.
```

When searching for counterexamples, QuickChick progressively tries larger and larger values for the size bound `n`, in order to explore larger and deeper part of the search space. Each generator can choose to interpret the size bound however it wants, and there is no enforced guarantee that generators pay attention to it at all; however, it is good practice to respect this bound when programming new generators.

Custom Generators Of course, we often need generators involving user-defined datatypes. To begin with, consider a simple enumeration of colors:

```
Inductive color := Red | Green | Blue.
```

To generate colors, we just need to pick one of the constructors `Red`, `Green` or `Blue`. To support this, QuickChick provide the `elements` combinator, which receives a list of elements of some type `A` and returns one of them uniformly at random.

```
elements : forall {A}, A -> list A -> G A
```

In Haskell's QuickCheck, the similar `elements` combinator raises an error on an empty list. Unlike Haskell however, Coq is a total language. Therefore, in QuickChick, `elements` takes an additional element that is returned when the list is empty. To avoid this inconvenience in the common case, QuickChick provides a shorthand notation `elems` for when the list is notempty. Thus, a generator for colors could simply be written as:

```
Definition genColor : G color :=
  elems [ Red ; Green ; Blue ].
```

For more complicated ADTs, QuickChick provides more combinators. We will showcase these using standard polymorphic binary trees; either `Leaf`s or `Node`s containing some payload of type `A` and two subtrees.

```
Inductive Tree A :=
| Leaf : Tree A
| Node : A -> Tree A -> Tree A -> Tree A.
```

The first useful generator combinator is `oneof`.

```
oneof : forall {A}, G A -> list (G A) -> G A
```

This combinator takes a default generator for some type `A` and a list of generators for the same type, and it picks one of the generators from the list uniformly at random (as long as the list is not empty, in which case it picks from the default generator). Just like with `elements`, QuickChick introduces a more convenient notation, `oneOf` to hide this default element.

The "obvious" first generator for trees that one might write is the following function `genTree`, which generates either a `Leaf` or else a `Node` whose subtrees are generated recursively (and whose payload is produced by a generator `g` for elements of type `A`).

```
Fixpoint genTree {A} (g : G A) : G (Tree A) :=
  oneOf [ ret Leaf ;;
          liftM3 Node g (genTree g) (genTree g)
        ].
```

At this point, Coq's termination checker is going to save us from shooting ourselves in the foot by disallowing this definition. Attempting to justify this fixpoint informally, one might first say that, at some point, the random generation will pick a `Leaf` so it will eventually terminate; a kind of probabilistic reasoning that the termination checker cannot understand. However, even informally, this reasoning is wrong: every time we choose to generate a `Node`, we create two separate branches that must both be terminated with a `Leaf`. From this, it is not hard to show that the expected size of the generated trees is actually infinite!

The solution is to use the standard "fuel" idiom that Coq users should be familiar with. We add an additional natural number `sz` as a parameter. We decrease this size in each recursive call, and when it reaches `O`, we always generate `Leaf`. Thus, the initial `sz` parameter serves as a bound on the depth of the tree.

```
Fixpoint genTreeSized {A} (sz : nat) (g : G A) : G (Tree A) :=
  match sz with
    | O => ret Leaf
    | S sz' =>
      oneOf
        [ ret Leaf ;
          liftM3 Node g (genTreeSized sz' g) (genTreeSized sz' g)
        ]
  end.
```

Now that we have a generator, let's generate some samples!

```
Sample (genTreeSized 3 (choose(0,3))).
```

```
                                                                    *output*
[ Leaf,
  Leaf,
  Node (3) (Node (0) (Leaf) (Leaf))
           (Node (2) (Leaf) (Node (3) (Leaf) (Leaf))),
  Leaf,
  Leaf,
  Leaf,
  Node (1) (Leaf) (Node (1) (Leaf) (Node (0) (Leaf) (Leaf))),
  Leaf,
  Node (3) (Leaf) (Leaf),
  Node (1) (Leaf) (Leaf),
  Leaf,
  Leaf,
  ... ]
```

While this generator succeeds in avoiding nontermination, we can see just by observing the result of `Sample` that there is a problem: `genTreeSized` produces way too many `Leaf`s! This is actually to be expected, since half the time we generate a `Leaf` right at the outset.

We can obtain bigger trees more often if we skew the distribution towards `Node`s using the most expressive QuickChick combinator, `frequency`.

```
frequency : forall {A}, G A -> list (nat * G A) -> G A
```

The `frequency` combinator, and its more convenient derived notation `freq`, take a list of generators, each tagged with a natural number that serves as the weight of that generator. For example, in the following generator, a `Leaf` will be generated $\frac{1}{\mathtt{sz}+1}$ of the time and a `Node` the remaining $\frac{\mathtt{sz}}{\mathtt{sz}+1}$ of the time.

```
Fixpoint genTreeSized' {A} (sz : nat) (g : G A) : G (Tree A) :=
  match sz with
    | O => ret Leaf
    | S sz' =>
        freq [ (1, ret Leaf) ;
               (sz, liftM3 Node g (genTreeSized' sz' g)
                                  (genTreeSized' sz' g))
             ]
  end.
```

Attempting to sample from this generator yields a much better looking distribution:

```
Sample (genTreeSized' 3 (choose(0,3))).
                                                              *output*
[ Node (3) (Node (1) (Node (3) (Leaf) (Leaf)) (Leaf))
           (Node (0) (Leaf) (Node (3) (Leaf) (Leaf))),
  Leaf,
  Node (2) (Node (1) (Leaf) (Leaf)) (Leaf),
  Node (0) (Leaf) (Node (0) (Node (2) (Leaf) (Leaf))
                            (Node (0) (Leaf) (Leaf))),
  Node (1) (Node (2) (Leaf) (Node (0) (Leaf) (Leaf)))
           (Leaf),
  Node (0) (Node (0) (Leaf) (Node (3) (Leaf) (Leaf)))
           (Node (2) (Leaf) (Leaf)),
  Node (1) (Node (3) (Node (2) (Leaf) (Leaf))
                     (Node (3) (Leaf) (Leaf)))
           (Node (1) (Leaf) (Node (2) (Leaf) (Leaf))),
  ...
]
```

Typeclasses for Generation QuickChick, in order to facilitate generator compositionality, offers two typeclasses, `GenSized` and `Gen`, that describe data that can be generated with sized and unsized generators respectively.

```
Class GenSized (A : Type) :=
  {
    arbitrarySized : nat -> G A
  }.

Class Gen (A : Type) :=
  {
    arbitrary : G A
  }.
```

We can easily provide a `GenSized` instance for `Trees` using the `genTreeSized'` generator we previously introduced:

```
Instance GenSizedTree {A} `{Gen A} : GenSized (Tree A) :=
  {| arbitrarySized n := genTreeSized' n arbitrary |}.
```

In this instance declaration, we assume that the type `A` already has a `Gen A` instance as a class constraint: instead of explicitly providing a generator for `A` as an argument, we allow Coq to fill it for us at every use site. [2] We then use the size argument `n` as input to `genTreeSized'`, and `arbitrary`, i.e. the typeclass method from `Gen A`, as the generator for the inner elements.

To go from a sized generator to an unsized one, QuickChick provides the `sized` combinator.

```
Definition sized {A : Type} (f : nat -> G A) : G A :=
  MkG (fun n r =>
         match f n with
           MkG g => g n r
       end).
```

[2] The notation `{ ... } is *implicit generalization*: unbound variables mentioned within are automatically bound in front of the binding where they occur.

All this `sized` combinator does is call the input parameterized generator `f` with the size argument drawn from the `G` monad, while taking care of the necessary internal random seed plumbing.

Armed with `sized`, we could write a `Gen` instance for trees.

```
Instance GenTree {A} `{Gen A} : Gen (Tree A) :=
  {
    arbitrary := sized arbitrarySized
  }.
```

Writing this instance for every single type would prove tedious rather quickly. However, nothing about it is `Tree`-specific! QuickChick provides a generic conversion function from `GenSized` to `Gen` instances.

```
Instance GenOfGenSized {A} `{GenSized A} : Gen A :=
  {
    arbitrary := sized arbitrarySized
  }.
```

Automation Finally, QuickChick comes with a derivation mechanism so that users don't have to write `GenSized` instances for every new custom datatype.

```
Derive GenSized for Tree.
```

```
                                                    *output*
GenSizedTree is defined
```

After executing the `Derive` vernacular command, QuickChick produces a generator that is identical to `genTreeSized`, up to alpha-conversion and frequency weights. To that end, QuickChick also provides a command, `QuickChickWeights`, for fine tuning the derived frequency weights.

```
QuickChickWeights [(Leaf, 1); (Node, 1)].
QuickChickWeights [(Leaf, 1); (Node, size)].
```

The first command will set the weights of both `Leaf` and `Node` to 1, as if we had used the `oneOf` combinator. The second will set the weight of `Node` to use the `size` parameter instead, just like our more efficient `genSizedTree'` generator.

2.3 Printers

The second component of property based testing we are going to explore is *printers*. In addition to the generation typeclasses, QuickChick provides a `Show` typeclass, just like Haskell's homonymous one.

```
Class Show A : Type :=
  {
    show : A -> string
  }.
```

QuickChick provides default instances for the most commonly used Coq datatypes: booleans, nats, integers, options, products, lists, etc. In fact, we have already used these instances to inspect the outcome of `Sample` in the previous section!

For user-defined datatypes it provides a derivation mechanism similar to that of generators. For example, we can obtain a generator for trees using the following vernacular command:

```
Derive Show for Tree.
```

```
                                                              *output*

ShowTree is defined
```

We can inspect the derived printer to see that it is relatively straightforward.

```
ShowTree = fun (A : Type) (_ : Show A) =>
{| show := fun x : Tree A =>
     let fix aux (x' : Tree A) : string :=
       match x' with
       | Leaf => "Leaf"
       | Node p0 p1 p2 =>
           "Node " ++
           smart_paren (show p0) ++ " " ++
           smart_paren (aux p1) ++ " " ++
           smart_paren (aux p2)
       end in
   aux x |}
```

The `show` method is defined as a fixpoint over the input type, here `Tree A`. If this input is a `Leaf`, we just return the string representation of the constructor's name; if it is a node, we print the constructors name, and then for each argument we either use the `show` method for that type (just like in `p0` above which has type `A`), or recursively call the fixpoint `aux`. Finally, we wrap these calls with calls to `smart_paren`, which just adds parentheses around a string, unless that string doesn't contain spaces, for prettier printing.

2.4 Shrinkers

A third, equally important component of property-based testing is shrinking. As we saw in the introduction with the `remove` example, counterexamples discovered by random testing tools are often large, containing a lot of noise that is irrelevant to the bug at hand. Shrinking is responsible for minimizing such counterexamples to smaller, minimal ones, to enable reasoning about the actual problems without additional clutter.

At its core, shrinking is just a greedy hill-climbing algorithm. Given a shrinking function `s` of type `A -> list A` and a value `x` of type `A` that is known to falsify some property `P`, QuickChick (lazily) tries `P` on all members of `s x` until it finds another counterexample. It then repeats this process starting from this new counterexample, until it reaches a point where `x` fails property `P` but every element of `s x` succeeds. This `x` is then a locally minimal counterexample. Naturally, this greedy algorithm only works if all elements of `s x` are strictly "smaller" than `x` for all `x`; that is, there should be some total order on the type of `x` such that `s` is strictly decreasing in this order.

Just like with generators and printers, QuickChick provides a typeclass, `Shrink`, for composing shrinkers in a simple fashion.

```
Class Shrink (A : Type) :=
  {
    shrink : A -> list A
  }.
```

Let's take a closer look at the default shrinking function for lists.

```
Fixpoint shrinkList {A : Type} (shr : A -> list A) (l : list A) :=
  match l with
  | [] => []
  | x :: xs =>
    xs ::
    List.map (fun xs' => x :: xs') (shrinkList shr xs) ++
    List.map (fun x'  => x' :: xs) (shr x)
  end
```

To shrink a list `l` with elements of some type `A`, we also need a shrinker for `A` (just like we needed a generator for `A` to generate trees of `A`). If the list is empty, there is nothing we can shrink to — the counterexample is already minimal. Otherwise, we can either drop the head of the list, recursively try to shrink the tail of the list, or try to shrink any one element of the list.

Just like with generators and printers, QuickChick provides instances for most of Coq's basic types, as well as support for automatically deriving shrinkers for any user-defined datatype:

```
Derive Shrink for Tree.
```

```
                                                              *output*
ShrinkTree is defined
```

2.5 Checkers

We can now use our generators, printers, and shrinkers to try and falsify properties, like the `delete_correct_bool` of the introduction:

```
Lemma delete_correct_bool (x : nat) (l : list nat) :
  ~~ (List.member x (delete x l)).
```

We would like to use QuickChick generators for `nat` and `list nat` to produce random inputs, check whether for each one `delete_correct_bool` returns `true` or

`false`, and then report shrunk versions of any counterexamples found. In other words, we want to use `delete_correct_bool` to build *a generator for test results*.

Results and Checkers To begin with, we need to define a type of results. We will start with a simplified version and build up increasingly more complex and useful ones in the course of this section.

```
Inductive Result := Success | Failure.
Derive Show for Result.
```

If we think of results as an enumerated type with two constructors, `Success` and `Failure`, then we can define the type of checkers to be a generator for `Result`:

```
Definition Checker := G Result.
```

A Checker embodies some way of performing a randomized test about the truth of a proposition, which will yield either `Success` (that is, the proposition survived this test) or `Failure` (that is, this test demonstrated that the proposition was false). Sampling a `Checker` many times causes many different tests to be performed.

To check `delete_correct_bool`, we'll need a way to build a `Checker` out of a function from `nat` to `list nat` to `bool`. Since we will in general need to build `Checker`s based on many different types, QuickChick defines a typeclass, `Checkable`, where an instance for `Checkable A` provides a way of converting an `A` into a `Checker`.

```
Class Checkable A :=
  {
    checker : A -> Checker
  }.
```

Instead of checking `delete_correct_bool` directly, let's start simpler and see how to build a `Checker` out of just a `bool`.

```
Instance checkableBool : Checkable bool :=
  {
    checker b := if b then ret Success else ret Failure
  }.
```

The boolean value true passes every test we might subject it to, while false fails all tests.

We can now sample these checkers!

```
Sample (checker true).
```

```
                                                              *output*
[Success, Success, Success, Success, Success,
 Success, Success, Success, Success, Success ]
```

```
Sample (checker false).
```

```
                                                              *output*
[Failure, Failure, Failure, Failure, Failure,
 Failure, Failure, Failure, Failure, Failure ]
```

Generating inputs We now know that the result of `delete_correct_bool` is `Checkable`. What we need is a way of taking a function returning a checkable thing and making the function itself checkable. We can easily do this, as long as the argument type of the function is something we know how to generate!

```
Definition forAll {A B : Type} `{Checkable B}
             (g : G A) (f : A -> B) : Checker :=
  a <- g ;;  checker (f a).
```

Armed with `forAll`, we can write a checker for the lemma we want to test, using `arbitrary` to generate both natural numbers and lists:

```
Definition delete_checker : Checker :=
  forAll arbitrary (fun x : nat =>
  forAll arbitrary (fun l : list nat =>
  delete_correct_bool x l)).
```

```
Sample delete_checker.
```

```
                                                              *output*
[Success, Success, Success, Success, Success,
 Failure, Success, Success, Success, Success ]
```

Great! The property fails! Unfortunately, there's one tiny problem: what are the tests that are failing? We can tell that the property is bad, but we can't see the counterexamples!

Showing counterexamples We can fix this by going back to the beginning and enriching the `Result` type to keep track of failing counterexamples.

```
Inductive Result :=
  | Success : Result
  | Failure : forall {A} `{Show A}, A -> Result.
```

The failure case for boolean checkers doesn't need to record anything except the Failure, so we put `tt` (the sole value of type `unit`) as the "failure reason".

```
Instance checkableBool : Checkable bool :=
  {
    checker b := if b then ret Success else ret (Failure tt)
  }.
```

The more interesting change lies in the `forAll` combinator. Here, we do have actual information to record in the failure case — namely, the argument that caused the failure.

```
Definition forAll {A B : Type} `{Show A} `{Checkable B}
                  (g : G A) (f : A -> B) : Checker :=
  a <- g ;;
  r <- checker (f a) ;;
  match r with
  | Success => ret Success
  | Failure b => ret (Failure (a,b))
  end.
```

Rather than just returning `Failure a`, we package up `a` together with `b`, which is the "reason" for the failure of `f a`. This allows us to write several `forAll`s in sequence and capture all of their results in a nested tuple.

Armed with the new `forAll` we can sample our checker for `delete` once again.

```
Sample delete_checker.
```

```
                                                              *output*
[Success, Success, Success, Success, Success,
 Failure : (17, ([42,0,4,-7,17,-6,-15,17,0,1,-13,8], tt)),
 Success, Success, Success, Success ]
```

Instead of using the rather awkward `Sample` command, we can instead use the `QuickChick` command. This command takes a `Checkable` input (a `Checker` like `delete_checker` is trivially `Checkable`), runs tests until a counterexample is found (or some predefined limit of successful runs is reached).

```
QuickChick delete_checker.
```

```
                                                              *output*
QuickChecking delete_checker...
*** Failed after 6 tests and 0 shrinks. (0 discards)
17
[42,0,4,-7,17,-6,-15,17,0,1,-13,8]
```

Adding back shrinking We run now again against the problem of random noise in the counterexamples. Even though we can now see the counterexample, it is too large to be practical: we cannot easily see why this property fails. That's where shrinking comes in. We can use the `forAllShrink` checker combinator, a variant of `forAll` that takes a shrinker as an additional argument, to define a better property...

```
Definition delete_checker' : Checker :=
  forAllShrink arbitrary shrink (fun x : nat =>
  forAllShrink arbitrary shrink (fun l : list nat =>
  delete_correct_bool x l)).
```

...and we can use `QuickChick` to test it:

```
QuickChick delete_checker'.
```

```
                                                              *output*
QuickChecking delete_checker'...
*** Failed after 6 tests and 11 shrinks. (0 discards)
0
[0,0]
```

At this point the bug is clearly identifiable: it occurs when we attempt to delete an element that is present twice in the list.

Putting it all together Now we've got pretty much all the basic machinery we need, but the way we write properties (using `forAllShrink` and explicitly providing generators and shrinkers) is a bit heavy. We can use a bit more typeclass magic to lighten things. First, for convenience, we package `Gen` and `Shrink` together into an `Arbitrary` typeclass that is a subclass of both.

```
Class Arbitrary (A : Type) `{Gen A} `{Shrink A}.
```

We can then provide a typeclass instance that automatically uses `Arbitrary` and `Show` instances to produce checkers for executable properties:

```
Instance testFun {A prop : Type} `{Show A} `{Arbitrary A}
                 `{Checkable prop} : Checkable (A -> prop) :=
{
  checker f := forAllShrink arbitrary shrink f
}.
```

Thus, we could directly check `delete_correct_bool` without explicitly annotating generators and shrinkers.

Inductive Specifications There still remains the question of testing specification in non-executable form. Indeed, Coq users usually write such specifications in propositional forms, which are not immediately `Checkable` like `bool`. For example, we saw in the earlier the following specification for our `delete` function:

```
Lemma delete_correct_prop : forall x l, ~ (In x (remove x l)).
```

To test such properties, QuickChick uses the `decidable` definition from ssreflect:

```
Print decidable.
```

```
                                                                *output*
decidable = fun P : Prop => {P} + {~P}
```

...where `{P} + {~P}` is an "informative disjunction" of `P` and `~P`. QuickChick wraps this in a typeclass of decidable propositions:

```
Class Dec (P : Prop) : Type :=
  {
    dec : decidable P
  }.
```

QuickChick also provides convenient notation `P?` that converts a decidable proposition `P` into a boolean expression.

```
Notation "P '?'" :=
  match (@dec P _) with
   | left _ => true
   | right _ => false
   end
  (at level 100).
```

Thus, a decidable `Prop` is not too different from a boolean, and we should be able to build a checker from that.

```
Instance checkableDec `{P : Prop} `{Dec P} : Checkable P :=
  {
    checker p := if P? then ret Success else ret (Failure tt)
  }.
```

This definition might look a bit strange since it doesn't use its argument `p`. The intuition is that all the information in `p` is already encoded in its type `P`!

We also need a counterpart to `testFun` for propositions:

```
Instance testProd {A : Type} {prop : A -> Type}
         `{Show A} `{Arbitrary A}
         `{forall x : A, Checkable (prop x)} :
  Checkable (forall (x : A), prop x) :=
  {| checker f :=
       forAllShrink arbitrary shrink (fun x => checker (f x))
  |}.
```

Finally, we can provide `Dec` instances for the `In` proposition (which is trivial, since such a lemma already exists in the Coq library) and directly `QuickCheck delete_correct_prop.`

Collecting statistics Earlier in this section we claimed that our first definition of `genTreeSized` produced "too many `Leaf`s". While looking at the result of `Sample` gives us a rough idea that something is going wrong, just observing a handful of samples cannot realistically provide statistical guarantees. This is where `collect`, another property combinator, comes in.

```
collect : forall {A P} `{Show A} `{Checkable P}, A -> P -> Checker
```

The `collect` combinator takes a `Checkable` proposition and returns a new `Checker` (intuitively, for the same proposition). On the side, it takes a value from some `Show`able type `A`, which it remembers internally (in an enriched variant of the `Result` structure that we saw above) so that it can be displayed at the end. For example, suppose we measure the size of `Tree`s like this:

```
Fixpoint size {A} (t : Tree A) : nat :=
  match t with
    | Leaf => O
    | Node _ l r => 1 + size l + size r
  end.
```

We could write a dummy property `treeProp` to check our generators and measure the size of generated trees.

```
Definition treeProp (g : nat -> G nat -> G (Tree nat)) n :=
  forAll (g n (choose (0,n))) (fun t => collect (size t) true).
```

QuickChecking this property results in the following statistics:

```
QuickChick (treeProp genTreeSized 5).
```

```
                                                              *output*
4947 : 0
1258 : 1
673 : 2
464 : 6
427 : 5
393 : 3
361 : 7
302 : 4
296 : 8
220 : 9
181 : 10
127 : 11
104 : 12
83 : 13
64 : 14
32 : 15
25 : 16
16 : 17
13 : 18
6 : 19
5 : 20
2 : 21
1 : 23

+++ OK, passed 10000 tests
```

We see that 62.5% of the tests are either `Leaf`s or empty `Node`s, while rather few

tests have larger sizes. Compare this with `genTreeSized'`:

```
QuickChick (treeProp genTreeSized' 5).
```

output

```
1624 : 0
571 : 10
564 : 12
562 : 11
559 : 9
545 : 8
539 : 14
534 : 13
487 : 7
487 : 15
437 : 16
413 : 6
390 : 17
337 : 5
334 : 1
332 : 18
286 : 19
185 : 4
179 : 20
179 : 2
138 : 21
132 : 3
87 : 22
62 : 23
20 : 24
17 : 25
+++ OK, passed 10000 tests
```

A lot fewer terms have small sizes, allowing us to explore larger terms.

2.6 Verifying QuickChick

In this chapter so far, we introduced QuickChick through a tutorial of its features, both adapted from Haskell's QuickCheck and novel Coq-specific ones. The uniqueness of QuickChick's setting gives rise to another option: instead of just using testing to facilitate interactive theorem proving, we can provide strong guarantees of the correctness of generators themselves [105].

Given a QuickChick generator `g` of type `G A` in the `G` monad, the function `semGen` assigns to it a (non-computable) set of outcomes (of type `A -> Prop`). That means that we can prove that `g` produces a desired set of outcomes `O` by showing that `semGen g` is extensionally equal to `O`, where sets (and extensional equality between them) have the following Coq representations:

```
Definition set T := T -> Prop.
Definition set_eq {A} (m1 m2 : set A) :=
  forall (a : A), m1 a <--> m2 a.
```

We write `<-->` as shorthand for `set_eq`.

QuickChick combinators come with precise, high-level specifications in terms of `semGen`. For an example, we can look at some combinators that appear in `GenSizedTree` in . For the simplest one, `ret`, the set of outcomes of `ret x` is just the singleton set {`x`}.

```
Lemma semReturn {A} (x : A) : semGen (ret x) <--> [set x].
```

A more complicated combinator we've encountered is `frequency`:

```
frequency: forall {A : Type}, G A -> list (nat * G A) -> G A
```

However, its specification is rather intuitive. Given a default generator `def` of type `G A` and a list `l` of pairs of natural numbers and generators, the set of outcomes of `frequency def l` depends on the natural weights chosen in the list. If there are no nonzero weights in the list (or if the list is empty), then the set of outcomes `semGen (frequency def l)` is the set of outcomes of the default generator (`semGen def`). Otherwise, it is the union of all the sets of outcomes of the nonzero-weighted generators in `l`.

```
Lemma semFrequency {A} (l : list (nat * G A)) (def : G A) :
  semGen (frequency def l) <-->
    let l' := [seq x <- l | x.1 != 0] in
    if l' is nil then semGen def else
      \bigcup_(x in l') semGen x.2.
```

Here, the notation `\bigcup_(i in A) F` denotes the union of all sets `F(i)` for the various `i` in the set `A`.

At this point, it would be tempting to give `bind` a similar intuitive specification as the `bind` of a probability/nondeterminism monad:

```
semGen (bind g f) <--> \bigcup_(a in semGen g) semGen (f a).
```

Unfortunately, this specification is wrong. The `G` monad (just like `Gen` in Haskell) is more than just a probability monad: it also provides *size* information that might be used by the generators. In the right hand side of the (wrong) spec, the size parameters passed to `g` and `f a` can differ, while `bind` calls both with the same one.

To facilitate reasoning in the presence of sizes QuickChick also provides specifications in terms of size; for most combinators the size variants just propagate size information inside.

```
Lemma semReturnSize A (x : A) (s : nat) :
  semGenSize (ret x) s <--> [set x].

Lemma semFreqSize {A} (l : list (nat * G A)) (def : G A)
      (size : nat) :
  semGenSize (frequency def l) size <-->
  let l' := [seq x <- l | x.1 != 0] in
  if l' is nil then semGenSize def size
  else \bigcup_(x in l') semGenSize x.2 size.
```

This approach allows us to give a (correct) specification for `bindGen` by threading the size parameter explicitly for both generators:

```
Lemma semBindSize :
  forall A B (g : G A) (f : A -> G B) (size : nat),
    semGenSize (bindGen g f) size <-->
    \bigcup_(a in semGenSize g size) semGenSize (f a) size.
```

Reasoning with low-level size information is very tedious. Fortunately, we can avoid it for a large class of generators, called *size-monotonic generators*, including the ones automatically derived by QuickChick. Size-monotonic generators produce larger sets of outcomes when given larger sizes as inputs:

```
Class SizeMonotonic {A} (g : G A) :=
  {
    monotonic :
      forall s1 s2, s1 <= s2 ->
      semGenSize g s1 \subset semGenSize g s2
  }.
```

For size-monotonic generators we have a simpler specification for `bindGen`:

```
Parameter semBindSizeMonotonic :
forall {A B} (g : G A) (f : A -> G B)
       `{SizeMonotonic _ g} `{forall a, SizeMonotonic (f a)},
  semGen (bindGen g f) <--> \bigcup_(a in semGen g) semGen (f a).
```

The `genTreeSized` generator we saw earlier has an implicit notion of size: `Leaf`s have size 0, while `Node`s have size 1 plus the maximum of the sizes of its left and right subtree. We generalize this notion to derive a `CanonicalSize` for arbitrary user-defined datatypes, with the same behavior.

For example, the canonical size of trees is derived automatically to be the following:

```
Instance sizeTree {A : Type} `{CanonicalSize A} :=
{|
   sizeOf := fun x : Tree A =>
       let fix aux_size (x' : Tree A) : nat :=
         match x' with
         | Leaf => 0
         | Node _ p1 p2 => S (max (aux_size p1) (aux_size p2))
         end in
       aux_size x
|}
```

To prove the monotonicity of our derived generators we first need to provide an induction principle of sorts, describing how to combine objects of specific sizes to create an object of an immediately larger size.

```
forall (size : nat) (A : Type),
Leaf
|: \bigcup_(f0 in (fun _ : A => true))
      \bigcup_(f1 in (fun f1 : Tree A => sizeOf f1 < size))
         \bigcup_(f2 in (fun f2 : Tree A => sizeOf f2 < size))
            [set Node f0 f1 f2]
<--> (fun x : Tree A => sizeOf x <= size)
```

The set of `Trees` `x` whose size is less than or equal to some specific natural number `size` consists of the base case, `Leaf`, as well as any `Node` whose left and right subtree are *strictly* smaller than `size`. This Lemma is straightforward, but extremely tedious to prove manually. Even worse, LTac automation seems to be very hard because it relies on knowing type information for various constructors appearing in its statement. Fortunately, the proof term is rather canonical and can be generated automatically! We will return again to this point of proving generators automatically in Chapter 5.

Acknowledgments

While the original iteration of QuickChick was written entirely by me during the first year of my studies, the now-available version is the product of labor by many amazing people. In particular, Zoe Paraskevopoulou and her master thesis advisor Catalin Hriţcu are the main authors of the proof framework for random generators. Moreover, Maxime Denes' familiarity with Coq and its internals was crucial in implementing the tool as a Coq plugin and maintaining it across versions. Finally, Benjamin Pierce has guided the development and progress of QuickChick towards success throughout its life.

The work presented in this chapter is adapted from various projects during the course of my dissertation. The main part of the tutorial is adapted from the QuickChick documentation and the lectures on "Property-based random testing with QuickChick" in the DeepSpec 2017 summer school, co-authored with Benjamin Pierce and Nicolas Koh. The proof-framework component of the tutorial is in turn adapted from our ITP 2015 paper "Foundational Property-Based Testing" [105], the majority of which exists thanks to Zoe Paraskevopoulou.

Chapter 3

Case Study: Information-Flow Control

In this chapter, we will present a case study on using random testing techniques to guide the design of a simple, low-level information-flow abstract machine. Secure information-flow control (IFC) is notoriously hard to achieve by careful design alone; the intricacies of the mechanisms involved, whether static [114] or dynamic [3, 4, 115, 42, 113, 130, 118, 64], make it hard to gain confidence in their correctness without formal proofs.

We will show that we can use QuickChick to speed the design of state-of-the-art IFC mechanisms by identifying bugs early in the design process, saving valuable time and effort. However, we will also draw attention to the complexity of the random generators involved, as well as the variety of subtle ways that one can introduce flaws in the testing code itself. Such flaws can lull a tester into a false sense of security, when confidence in the correctness of a program is ill founded.

3.1 Stack Machine

Before presenting the abstract stack machine, we need to introduce some notation: if xs is a list and $0 \leqslant j < |xs|$, then $xs(j)$ selects the j^{th} element of xs and $xs[j := x]$ produces the list that is like xs except that the j^{th} element is replaced by x.

3.1.1 Abstract Machine

In a (fine-grained) dynamic IFC system [3, 118, 64, 115, 62, 13, 6] security levels (called labels) are attached to runtime values and propagated during execution, enforcing the constraint that information derived from secret data does not leak to untrusted processes or to the public network. Each value is protected by an individual IFC label representing a security level (e.g., secret or public).

Instead of bare integers, the basic data items in our abstract machine are *labeled integers* of the form $n@\ell$, where n is an integer and ℓ is a *label*:

$$\ell ::= L \mid H$$

We read L as "low" (public) and H as "high" (secret). Additionally, we order labels by $L \sqsubseteq H$ and write $\ell_1 \vee \ell_2$ for the *join* (least upper bound) of ℓ_1 and ℓ_2.

The instructions of our simple stack machine are unsurprising.

$$\begin{aligned} \mathit{Instr} ::= {} & \mathit{Push}\ n@\ell \mid \mathit{Pop} \mid \mathit{Load} \mid \mathit{Store} \mid \mathit{Add} \mid \mathit{Noop} \\ \mid {} & \mathit{Jump} \mid \mathit{Call}\ n\ k \mid \mathit{Return} \mid \mathit{Halt} \end{aligned}$$

The argument to *Push* is a labeled integer to be pushed on the stack and the numeric arguments to *Call* the number of integers that should be passed or returned (0 or 1 in the latter case). To account for stack frames, each stack element e can either be a labeled integer $n@\ell$ or a *return address*, marked R, recording the *pc* from which the corresponding *Call* was made, as well as the number of arguments to be returned.

Finally, a machine state S is a 4-tuple, written $\boxed{pc \mid s \mid m \mid i}$, consisting of a program counter *pc* (an integer), a stack s, a memory m (a list of labeled integers), and an instruction memory i (a list of instructions). Since i cannot change during execution we will often write just $\boxed{pc \mid s \mid m}$ for the varying parts of the machine state. The set of *initial* states of this machine, *Init*, contains states of the form $\boxed{0 \mid [\,] \mid m_0 \mid i}$, where m_0 can be of any length and contains only $0@L$. We use *Halted* to denote the set of halted states of the machine, i.e., $i(pc) = \mathit{Halt}$.

3.1.2 Noninterference

We then need to define what it means for such a machine to be "secure" using a standard notion of termination-insensitive noninterference [114, 64, 3, 6]; we call it *end-to-end noninterference* (or *EENI*) to distinguish it from the stronger notions we will introduce later on. In EENI we directly encode the intuition that secret inputs should not influence public outputs. By secret inputs we mean integers labeled H in the initial state (because of the simplicity of our initial states, such labeled integers can appear only in instruction memories); by public outputs we mean integers labeled L in a halted state.

More precisely, EENI states that for any two executions starting from initial states that are *indistinguishable to a low observer* (or just *indistinguishable*) and ending in halted states S_1 and S_2, the final states S_1 and S_2 are also indistinguishable. We write $S \Downarrow S'$ to denote an execution starting from S and ending in S'. Intuitively, two states are indistinguishable if they differ only in integers labeled H. To make this formal, we define an equivalence relation on states compositionally from equivalence relations over their components.

Definition 3.1.2.1.

- Two labeled integers $n_1@\ell_1$ and $n_2@\ell_2$ are said to be *indistinguishable*, written $n_1@\ell_1 \approx n_2@\ell_2$, if either $\ell_1 = \ell_2 = H$ or else $n_1 = n_2$ and $\ell_1 = \ell_2 = L$.
- Two instructions i_1 and i_2 are indistinguishable if they are the same, or if $i_1 = \mathit{Push}\ n_1@\ell_1$, and $i_2 = \mathit{Push}\ n_2@\ell_2$, and $n_1@\ell_1 \approx n_2@\ell_2$.
- Two return addresses $R(n_1, k_1)@\ell_1$ and $R(n_2, k_2)@\ell_2$ are indistinguishable if either $\ell_1 = \ell_2 = H$ or else $n_1 = n_2$, $k_1 = k_2$ and $\ell_1 = \ell_2 = L$.
- Two lists (memories, stacks, or instruction memories) xs and ys are indistinguishable if they have the same length and $xs(i) \approx ys(i)$ for all i such that $0 \leqslant i < |xs|$.

Definition 3.1.2.2. Machine states $S_1 = \boxed{pc_1 \mid s_1 \mid m_1 \mid i_1}$ and $S_2 = \boxed{pc_2 \mid s_2 \mid m_2 \mid i_2}$ are *indistinguishable with respect to memories*, written $S_1 \approx_{mem} S_2$, if $m_1 \approx m_2$ and $i_1 \approx i_2$.

Definition 3.1.2.3. A machine semantics is *end-to-end noninterfering* with respect to some sets of states *Start* and *End* and an indistinguishability relation $\approx$, written $\text{EENI}_{Start,End,\approx}$, if for any $S_1, S_2 \in Start$ such that $S_1 \approx S_2$ and such that $S_1 \Downarrow S_1'$, $S_2 \Downarrow S_2'$, and $S_1', S_2' \in End$, we have $S_1' \approx S_2'$.

We take $\text{EENI}_{Init,Halted,\approx_{mem}}$ as our baseline security property; i.e., we only consider executions starting in initial states and ending in halted states, and we use indistinguishability with respect to memories [1]. The EENI definition above is, however, more general, and we will consider other instantiations of it later.

3.1.3 Operational Semantics

The final task is to enrich standard rules for the step function to take information-flow labels into account. For most of the rules, there are multiple plausible ways to do this and there are a lot of opportunities for subtle mistakes. In "Testing Noninterference, Quickly", we illustrated a test-driven development approach: we first proposed a naive set of rules and then used counterexamples generated using QuickChick and custom generation and shrinking techniques (which we will describe in later sections) to identify and help repair mistakes until no more could be found. Here, we will demonstrate this methodology using a progression of increasingly more refined rules for a single instruction (*Store*) and present the final version of the rules directly, as discovered by random testing and verified in the Coq proof assistant. This sound and permissive set of rules (and the single-step reduction relation on machine states, written $S \Rightarrow S'$, it gives rise to) appears in Figure 3.1.3.

A *Noop* simply increments the program counter by 1, leaving the rest of the machine unaffected (all non-control-flow instructions have the same effect on the program counter and we will omit it from the explanation). A *Push* $n@\ell$ instruction

[1] At this point we have a choice as to how much of the state we want to consider observable; we choose (somewhat arbitrarily) that the observer can only see the data and instruction memories, but not the stack or the *pc*. Other choices would give the observer either somewhat more power—e.g., we could make the stack observable—or somewhat less—e.g., we could restrict the observer to some designated region of "I/O memory," or extend the architecture with I/O instructions and only observe the traces of inputs and outputs [6]

$$\frac{i(pc) = \mathit{Noop}}{\boxed{pc@\ell_{pc}}\,\boxed{s}\,\boxed{m} \Rightarrow \boxed{pc{+}1@\ell_{pc}}\,\boxed{s}\,\boxed{m}} \quad (\textsc{Noop})$$

$$\frac{i(pc) = \mathit{Push}\ n@\ell}{\boxed{pc@\ell_{pc}}\,\boxed{s}\,\boxed{m} \Rightarrow \boxed{pc{+}1@\ell_{pc}}\,\boxed{n@\ell : s}\,\boxed{m}} \quad (\textsc{Push})$$

$$\frac{i(pc) = \mathit{Add}}{\boxed{pc@\ell_{pc}}\,\boxed{n_1@\ell_1 : n_2@\ell_2 : s}\,\boxed{m} \Rightarrow \boxed{pc{+}1@\ell_{pc}}\,\boxed{(n_1{+}n_2)@(\ell_1 \vee \ell_2) : s}\,\boxed{m}} \quad (\textsc{Add})$$

$$\frac{i(pc) = \mathit{Pop}}{\boxed{pc@\ell_{pc}}\,\boxed{n@\ell_n : s}\,\boxed{m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc}}\,\boxed{s}\,\boxed{m}} \quad (\textsc{Pop})$$

$$\frac{i(pc) = \mathit{Load} \qquad m(p) = n@\ell_n}{\boxed{pc@\ell_{pc}}\,\boxed{p@\ell_p : s}\,\boxed{m} \Rightarrow \boxed{pc{+}1@\ell_{pc}}\,\boxed{n@(\ell_n \vee \ell_p) : s}\,\boxed{m}} \quad (\textsc{Load})$$

$$\frac{i(pc) = \mathit{Store} \quad m(p) = n'@\ell'_n \quad \ell_p \vee \ell_{pc} \sqsubseteq \ell'_n \quad m' = m[p := n@(\ell_n \vee \ell_p \vee \ell_{pc})]}{\boxed{pc@\ell_{pc}}\,\boxed{p@\ell_p : n@\ell_n : s}\,\boxed{m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc}}\,\boxed{s}\,\boxed{m'}} \quad (\textsc{Store})$$

$$\frac{i(pc) = \mathit{Jump}}{\boxed{pc@\ell_{pc}}\,\boxed{n@\ell_n : s}\,\boxed{m} \Rightarrow \boxed{n@(\ell_n \vee \ell_{pc})}\,\boxed{s}\,\boxed{m}} \quad (\textsc{Jump})$$

$$\frac{i(pc) = \mathit{Call}\ k\ k' \qquad k' \in \{0,1\} \qquad ns = n_1@\ell_1 : \ldots : n_k@\ell_k}{\boxed{pc@\ell_{pc}}\,\boxed{n@\ell_n : ns : s}\,\boxed{m} \Rightarrow \boxed{n@(\ell_n \vee \ell_{pc})}\,\boxed{ns : R(pc{+}1, k')@\ell_{pc} : s}\,\boxed{m}} \quad (\textsc{Call})$$

$$\frac{i(pc) = \mathit{Return} \qquad ns = n_1@\ell_1 : \ldots : n_{k'}@\ell_{k'} \qquad ns_{pc} = n_1@(\ell_1 \vee \ell_{pc}) : \ldots : n_{k'}@(\ell_{k'} \vee \ell_{pc})}{\boxed{pc@\ell_{pc}}\,\boxed{ns : ns' : R(n, k')@\ell : s}\,\boxed{m} \Rightarrow \boxed{n@\ell}\,\boxed{ns_{pc} : s}\,\boxed{m}} \quad (\textsc{Return})$$

Figure 3.1: Single-step Reduction Rules

adds $n@\ell$ to the stack. An *Add* instruction adds the two top elements of the stack and pushes the result. The label of the new element is equal to the join of the labels of the values added, as is standard in information-flow control. A *Pop* just drops the top element of the stack, which must not be a return frame. A *Load* instruction takes a value $p@\ell_p$ off the stack, finds the element at the p-th location of the memory and returns it with its label tainted by ℓ_p. A *Store* instruction takes two elements off the stack, uses the value p of the first as an address and updates the p-th location of the memory to contain the second. The (rather complicated) IFC content will be discussed shortly. *Jump* updates the pc to the address pointed by the top element of the stack. *Call* takes an address off the stack, updates the pc and adds a stack frame separator (marked R) in the stack. Finally, *Return* looks for a stack frame $R(n, k')@\ell$, updates the pc address to n and keeps the top k' elements from the stack (as return values of the corresponding call).

3.1.4 Test Driven Development

To showcase our development methodology, let's consider a version of the *Store* rule without the complicated information-flow checks.

$$\frac{i(pc) = \mathit{Store} \qquad m' = m[p := n@\ell_n]}{\boxed{pc \mid p@\ell_p : n@\ell_n : s \mid m} \Rightarrow \boxed{pc{+}1 \mid s \mid m'}} \qquad \text{(STORE-1)}$$

In the rule above, the label of the element stored in the memory remains unchanged and no checks are done. Using QuickChick, we quickly discover the following counterexample to this formulation.

$$i = \left[\ \mathit{Push}\ 1@L, \mathit{Push}\ \tfrac{0}{1}@H, \mathit{Store}, \mathit{Halt}\ \right]$$

pc	m	s	$i(pc)$
0	$[0@L, 0@L]$	$[\,]$	$\mathit{Push}\ 1@L$
1	$[0@L, 0@L]$	$[1@L]$	$\mathit{Push}\ \frac{0}{1}@H$
2	$[0@L, 0@L]$	$\left[\frac{0}{1}@H, 1@L\right]$	Store
3	$\left[\frac{1}{0}@L, \frac{0}{1}@L\right]$	$[\,]$	Halt

The first line of the figure is the counterexample itself: a pair of four-instruction programs, differing only in the constant argument of the second *Push*. The first program pushes $0@H$, while the second pushes $1@H$ (since the labels are high, these two labeled integers are indistinguishable). We display the two programs, and the other parts of the two machine states, in a "merged" format. Pieces of data that are the same between the two machines are written just once; at any place where the two machines differ, the value of the first machine is written above the value of the second machine, separated by a horizontal line.

The rest of the figure shows what happens when we run this program. On the first step, the *pc* starts out at 0; the memory, which has two locations, starts out as $[0@L, 0@L]$; the stack starts out empty; and the next instruction to be executed ($i(pc)$) is *Push* $1@L$. On the next step, this labeled integer has been pushed on the stack and the next instruction is either *Push* $0@H$ or *Push* $1@H$; one or the other of these labeled integers is pushed on the stack. On the next, we *Store* the second stack element ($1@L$) into the location pointed to by the first (either $0@H$ or $1@H$), so that now the memory contains $1@L$ in either location 0 or location 1 (the other location remains unchanged, and contains $0@L$). At this point, both machines halt.

This pair of execution sequences shows that EENI fails: in the initial state, the two programs are indistinguishable to a low observer (their only difference is labeled H). However, in the final states the memories contain different integers at the same location, both of which are labeled L.

Thinking about this counterexample, it soon becomes apparent what went wrong with the *Store* instruction: since pointers labeled H are allowed to vary between the two runs, it is not safe to store a low integer through a high pointer. One simple but draconian fix is simply to stop the machine if it tries to perform such a store (i.e., we could add the side condition $\ell_p = L$ to the rule). A more permissive option is to allow the store to take place, but require it to taint the stored value with the label on the pointer:

$$\frac{i(pc) = \mathit{Store} \qquad m' = m[p := n@(\ell_n \vee \ell_p)]}{\boxed{pc}\,\boxed{p@\ell_p : n@\ell_n : s}\,\boxed{m} \Rightarrow \boxed{pc{+}1}\,\boxed{s}\,\boxed{m'}} \qquad \text{(STORE-2)}$$

Sadly, the next counterexample shows that this rule is still not quite good enough.

$$i = \left[\ \mathit{Push}\ 0@L, \mathit{Push}\ \tfrac{0}{1}@H, \mathit{Store}, \mathit{Halt}\ \right]$$

pc	m	s	$i(pc)$
0	$[0@L, 0@L]$	$[\,]$	$\mathit{Push}\ 0@L$
1	$[0@L, 0@L]$	$[0@L]$	$\mathit{Push}\ \frac{0}{1}@H$
2	$[0@L, 0@L]$	$\left[\frac{0}{1}@H, 0@L\right]$	Store
3	$\left[0@\frac{H}{L}, 0@\frac{L}{H}\right]$	$[\,]$	Halt

This counterexample is quite similar to the first one, but it illustrates a more subtle point: our definition of noninterference allows the observer to distinguish between final memory states that differ only in their *labels*.[2] Since the STORE-2 rule taints the label of the stored integer with the label of the pointer, the fact that the *Store* changes different locations is visible in the fact that a label changes from L to H on a different memory location in each run. To avoid this issue, we adopt the "no sensitive upgrades" rule [128, 3], which demands that the label on the current contents of a memory location being stored into are above the label of the pointer used for the store —i.e., it is illegal to overwrite a low value via a high pointer (and trying to do so results in a fatal failure).

$$\frac{i(pc) = \mathit{Store} \qquad m(p) = n'@\ell'_n \qquad \ell_p \sqsubseteq \ell'_n \qquad m' = m[p := n@(\ell_n \vee \ell_p)]}{\boxed{pc}\,\boxed{p@\ell_p : n@\ell_n : s}\,\boxed{m} \Rightarrow \boxed{pc{+}1}\,\boxed{s}\,\boxed{m'}} \quad \text{(STORE-3)}$$

[2] See the first clause of Definition 3.1.2.1. One might imagine that this could be fixed easily by changing the definition so that whether a label is high or low is not observable—i.e., $n@L \approx n@H$ for any n. Sadly, this is known not to work [113, 42]. (QuickChick can also find a counterexample)

Moving on, there are *still* problems with rule STORE-3; it doesn't take the pc label into account.

$$i = \left[\begin{array}{l} \mathit{Push}\ \frac{3}{6}@H, \mathit{Call}\ 0\ 0, \mathit{Halt}, \mathit{Push}\ 1@L, \mathit{Push}\ 0@L, \\ \mathit{Store}, \mathit{Return} \end{array} \right]$$

pc	m	s	$i(pc)$
0	$[0@L]$	$[\,]$	$\mathit{Push}\ \frac{3}{6}@H$
1	$[0@L]$	$\left[\frac{3}{6}@H\right]$	$\mathit{Call}\ 0\ 0$
Machine 1 continues...			
3	$[0@L]$	$[R(2,0)@L]$	$\mathit{Push}\ 1@L$
4	$[0@L]$	$[1@L, R(2,0)@L]$	$\mathit{Push}\ 0@L$
5	$[0@L]$	$[0@L, 1@L, R(2,0)@L]$	Store
6	$[1@L]$	$[R(2,0)@L]$	Return
2	$[1@L]$	$[\,]$	Halt
Machine 2 continues...			
6	$[0@L]$	$[R(2,0)@L]$	Return
2	$[0@L]$	$[\,]$	Halt

This counterexample shows we need to be careful about stores in high contexts. We change the rule to taint the value written in memory with the current *pc* label:

$$\frac{i(pc) = \mathit{Store} \quad m(p) = n'@\ell'_n \quad \ell_p \sqsubseteq \ell'_n \quad m' = m[p := n@(\ell_n \vee \ell_p \vee \ell_{pc})]}{\boxed{pc@\ell_{pc}}\boxed{p@\ell_p : n@\ell_n : s}\boxed{m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc}}\boxed{s}\boxed{m'}} \text{ (STORE-4)}$$

This eliminates the current counterexample; QuickChick then uncovers a very similar one in which the *labels* of values in the memories differ between the two machines. The usual way to prevent this problem is to extend the no-sensitive-upgrades check so that low-labeled data cannot be overwritten in a high context [128, 3]. This leads to the correct rule for *Store*, first seen in Figure 3.1.3.

$$\frac{i(pc) = \mathit{Store} \quad m(p) = n'@\ell'_n \quad \ell_p \vee \ell_{pc} \sqsubseteq \ell'_n \quad m' = m[p := n@(\ell_n \vee \ell_p \vee \ell_{pc})]}{\boxed{pc@\ell_{pc}}\boxed{p@\ell_p : n@\ell_n : s}\boxed{m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc}}\boxed{s}\boxed{m'}} \text{ (STORE)}$$

3.2 Testing

While the counterexample progression in the previous section clearly shows the benefits of using random testing to design a complex system, such an approach does not come without effort. Indeed, to present such polished counterexamples we put a lot of work into generating, shrinking and pretty printing abstract machines. Moreover, for more efficient testing, we needed to strengthen the property under test as the machine grew more complex. The rest of this section is devoted to describing the various techniques developed for these purposes. More importantly, we interleave *critiques* of these techniques and sketch directions where research opportunities arise.

3.2.1 Generation Techniques

To test our design we need to construct random indistinguishable *pairs* of machine states. If we were to generate two random states independently, the chance of them being indistinguishable is minuscule, since they would have to coincide in *all* "low" places. The obvious solution to this problem is to generate one abstract machine first and then to create the second machine by randomly *varying* the "high" parts of the first one.

At this point, and every time when we write a custom generator satisfying some precondition (here indistinguishability), we need to ensure that our generation process is complete with respect to the predicate: if there exist indistinguishable pairs of states that cannot be generated, we might not be able to find certain bugs! This will be a growing concern throughout this section: while generating variations of initial states is trivial, as the complexity of our generation strategies grows, so does the possibility of incompleteness.

The next aspect of generation we need to tackle is the generation of instructions. A naive first approach by generating instructions independently and uniformly doesn't work: most of the time, machines fail to reach a halted state as required by EENI and the generated inputs are therefore discarded. In fact, by collecting statistics about the various executions, we learned that the average execution length is less than half a step! Clearly, such short runs cannot lead to effective bug

finding. We need to fine tune the distribution of instructions to achieve longer, more interesting runs.

Further statistics show that the most common reason for early termination is stack underflows. Indeed, starting from the empty stack of an initial state, unless the next instruction is a *Push*, *Halt* or a *Noop*, execution will fail. To counter this problem we leverage the `frequency` combinator we introduced in the previous chapter. As a reminder, `frequency` (with type `list (nat * G A) -> G A`) takes a list of pairs of natural numbers and generators and picks a generator at random, based on the distribution induced by the weights. By increasing the weight of *Push* instructions, we reduce stack underflows; by increasing the likelihood of *Halt*, fewer tests fail to satisfy the termination condition of EENI; by increasing *Store*s, we ensure any information-flow violations become observable by appearing in the memory. This weighted distribution of instructions leads to longer runs (average 2.69 steps) and lowers discard rates (by 16%), which translates to an order of magnitude better testing performance in terms of mean time to find a failing case (MTTF) for all bugs that were found.

We further improve the MTTF by generating useful instruction sequences *together*: for example, pushing an integer followed by a *Load* or pushing two integers followed by a *Store*. Another order-of-magnitude improvement can be attained by skewing the distribution of generated integers towards valid addresses (leading to fewer out-of-bounds errors).

However, the most important generation technique developed was *generation by execution*: we generate a single instruction (or a small sequence) from a restricted subset that does not cause the machine to crash in the current state. For instance, we never generate an *Add* instruction if there are not at least two elements in the stack to pop and operate on. Afterwards, we execute the generated instruction, reach a new state and repeat. Due to control flow, it is possible to reach a state where the next instruction has already been generated. In that case, we just keep execution going until the next instruction hasn't been generated (or until we reach a loop-avoiding predetermined cutoff). Finally, to ensure that generated machines successfully terminate we increase the likelihood of *Halt*s the more instructions we generate. The combination of all generation strategies leads to consistently finding

all but one injected bugs with a (geometric) average MTTF of 334.6 seconds.

3.2.2 Strengthening the Property

EENI While $\text{EENI}_{Init,Halted,\approx_{mem}}$ is the property we ultimately care about, it does not lend itself to efficient testing. Much like when proving noninterference we would come up with stronger, potentially inductive, specifications that imply EENI and are easier to prove, the same is true for testing: stronger properties can be much better at finding bugs.

A first observation is that information flows often appear early in counterexamples in the form of a low variation in the stack or the *pc*, but then it is necessary to store the leak into the memory to make it observable by our property. By redefining indistinguishability to take into account the entire machine state, we can obtain shorter counterexamples (that are easier to find):

Definition 3.2.2.1. Machine states $S_1 = \boxed{pc_1 \mid s_1 \mid m_1 \mid i_1}$, $S_2 = \boxed{pc_2 \mid s_2 \mid m_2 \mid i_2}$ are indistinguishable with respect to entire low states, written $S_1 \approx_{low} S_2$, if either $\ell_{pc_1} = \ell_{pc_2} = H$ or else $\ell_{pc_1} = \ell_{pc_2} = L$, $m_1 \approx m_2$, $i_1 \approx i_2$, $s_1 \approx s_2$, and $pc_1 \approx pc_2$.

A second, dual observation is that all counterexamples begin by pushing elements onto the stack before exploiting some information-flow bug to leak a secret. This is necessary since initial states in our stack machine begin with an empty stack! By generating *quasi-initial* states, containing arbitrary (but indistinguishable with respect to $\approx_{low}$) stacks in addition to memories, we can significantly improve the average MTTF. However, this approach doesn't come without a price. When generating such quasi-initial states, there is no guarantee that such a state is actually *reachable* from an initial state. In principle, that means that QuickChick could report spurious problems that cannot actually arise in any real situation. In general, we can address such problems by carefully formulating *invariants* of reachable states and ensuring that we generate quasi-initial states satisfying them. In practice, though, for this extremely simple machine we did not encounter any spurious counterexamples (that was not the case for an extended, more complicated register machine that we tried afterwards!).

Thus, by instantiating EENI appropriately, we obtain a stronger property $\text{EENI}_{QInit, Halted \cap Low, \approx_{low}}$ that finds all bugs much faster, with a geometric average MTTF of 46.48 seconds, an order of magnitude less than the baseline of 334.6.

LLNI Making the full state observable and starting from quasi-initial states significantly improved testing performance. However, we can get even better results by moving to a yet stronger noninterference property. The intuition is that EENI generates machines and runs them for a long time, but it only compares the final states, and only when both machines successfully halt; these preconditions lead to rather large discard rates. On the other hand, comparing *intermediate* states as well and reporting a bug as soon as intermediate states are distinguishable can lead to yet shorter and easier-to-find counterexamples. While the *pc* is high, the two machines may be executing different instructions, so their states will naturally differ; we ignore these states and require only that low execution states are pointwise indistinguishable. We call this new property *low-lockstep noninterference* (or *LLNI*). Benchmarking LLNI in the same bugs reveals a further increase in performance, with an impressive MTTF of only 7.69 seconds!

SSNI Still, there is more room for improvement! LLNI essentially checks the following invariant: if two low indistinguishable machines take a step and remain low, then they stay indistinguishable. The drawback is when machines are in a high state they are allowed to differ arbitrarily. In a noninterference proof, an inductive invariant would have to guarantee that after after the machines go back to a low state, they would be low indistinguishable. In our development, this stronger invariant gives rise to the *single-step noninterference* property (SSNI).

Definition 3.2.2.2. A machine semantics is *single-step noninterfering* with respect to an indistinguishability relation $\approx$ (written $\text{SSNI}_{\approx}$) if the following conditions (often called *unwinding conditions*) are satisfied:

1. For all $S_1, S_2 \in Low$, if $S_1 \approx S_2$, $S_1 \Rightarrow S_1'$, and $S_2 \Rightarrow S_2'$, then $S_1' \approx S_2'$;
2. For all $S \notin Low$ if $S \Rightarrow S'$ and $S' \notin Low$, then $S \approx S'$;

3. For all $S_1, S_2 \notin \mathit{Low}$, if $S_1 \approx S_2$, $S_1 \Rightarrow S_1'$, $S_2 \Rightarrow S_2'$, and $S_1', S_2' \in \mathit{Low}$, then $S_1' \approx S_2'$.

Note that SSNI talks about completely arbitrary states, not just initial or quasi-initial ones.

The definition above is parametric in the indistinguishability relation used. Finding the right relation can take some work, just like in proofs! Fortunately, QuickChick can help with this process as well. Low indistinguishability ($\approx_{low}$) is too weak and QuickChick can easily find counterexamples to condition 3, e.g., by choosing two indistinguishable machine states with $i = [\mathit{Return}]$, $pc = 0$, and $s = \left[R(\frac{0}{1}, 0)@L\right]$; after a single step the two machines have distinguishable pcs 0 and 1, respectively. On the other hand, treating high states exactly like low states in the indistinguishability relation is too strong. In this case QuickChick finds counterexamples to condition 2, e.g., a single machine state with $i = \left[\ \mathit{Pop}\ \right]$, $pc = 0$, and $s = [0@L]$ steps to a state with $s = []$, which would not be considered indistinguishable.

These counterexamples show that indistinguishable high states can have different pcs and can have completely different stack frames at the top of the stack. So all we can require for two high states to be equivalent is that their memories and instruction memories agree and that the parts of the stacks below the topmost low return address are equivalent. This is strong enough to ensure condition 3.

Definition 3.2.2.3. Machine states $S_1 = \boxed{pc_1 \mid s_1 \mid m_1 \mid i_1}$, $S_2 = \boxed{pc_2 \mid s_2 \mid m_2 \mid i_2}$ are *indistinguishable with respect to whole machine states*, written $S_1 \approx_{full} S_2$, if $m_1 \approx m_2$, $i_1 \approx i_2$, $\ell_{pc_1} = \ell_{pc_2}$, and additionally

- if $\ell_{pc_1} = L$ then $s_1 \approx s_2$ and $pc_1 \approx pc_2$, and
- if $\ell_{pc_1} = H$ then $\mathit{cropStack}\ s_1 \approx \mathit{cropStack}\ s_2$.

The *cropStack* helper function takes a stack and removes elements from the top until it reaches the first low return address (or until all elements are removed).

Using $\text{SSNI}_{\approx_{full}}$ for testing, even with arbitrary starting states, performs very well (12.87 seconds MTTF). At that point, it felt natural that since machines are

executed only for one step, we could get away with very small states. Indeed, after fine-tuning the resulting distribution we got a MTTF average of less than 0.5 seconds! Once again however, messing with the generation (this time to only produce tiny states) is not without risks. For example, we originally only created instruction memories of size 1 (the single instruction to be executed). Unfortunately, that is not enough to exhibit bugs where the secrets leaked concern instruction memory pointers: without two different instruction memory locations all valid pointers are equal!

MSNI When optimizing the generation for SSNI, we must be extremely cautious to avoid ruling out useful parts of the state space. Since SSNI operates by executing a machine state for a *single* step to check the invariant, generating the *complete* state space of pairs of indistinguishable machines becomes very important.

Comparing LLNI and SSNI with respect to their efficiency in testing, we identify an interesting tradeoff. On the one hand, a significant limitation of LLNI is that bugs that appear when the *pc* is high are not detected immediately, but only after the *pc* goes back low, if ever. One example is the STORE-4 buggy rule presented earlier, where we do not check whether the *pc* label flows to the label of the memory cell. On such bugs LLNI demonstrates orders of magnitude worse bug-finding efficiency. On the other hand, SSNI is significantly less robust with respect to starting state generation. If we do not generate every valid starting state, then SSNI will not test executions starting in the missing states, since it only executes one instruction. LLNI avoids this problem as long as all valid states are potentially reachable from the generated starting states.

These observations lead us to formulate one final property that combines the advantages of both LLNI and SSNI: : *multi-step noninterference (MSNI)*. The formal definition of MSNI is given in the journal version of the paper [66]. Informally, we start from an arbitrary pair of indistinguishable machine states and we check the SSNI invariant along a whole execution trace. Using generation by execution with fine-tuned instruction frequencies for this property leads to an efficiency that is on par with the better of SSNI or LLNI, uncovering IFC violations as soon as they appear; at the same time, unlike SSNI, MSNI is robust against faulty generation.

3.3 Experiences from Extending the Machine

A natural question that arose from this line of work is whether our testing methodology scales to larger, more realistic machines. To answer that question we extended the machine to include registers, as well as advanced IFC features such as a richer lattice of first-class labels and dynamically allocated memory with mutable labels. Presenting all the features of that machine here is out of scope of this thesis; the interested reader is referred to the journal version of the "Testing Noninterference, Quickly" paper [66]. However, two specific extensions will allow us to continue the discussion of custom generators: using a larger label lattice and parameterizing the IFC rules in a rule table.

3.3.1 Decoupling of Generators and Predicates

In the extended machine, we moved from a two-point lattice for labels to an arbitrary lattice. For the purposes of this section, we can restrict our attention to a four-element diamond lattice:

$$\ell ::= L \mid M_1 \mid M_2 \mid H$$

where $L \sqsubseteq M_1$, $L \sqsubseteq M_2$, $M_1 \sqsubseteq H$, and $M_2 \sqsubseteq H$. The labels M_1 and M_2 are incomparable. With this richer lattice, our definition of "low" and "high" becomes relative to an arbitrary observer label ℓ: we call some label ℓ' low with respect to ℓ if $\ell' \sqsubseteq \ell$ and high otherwise.

In the new setting, a correct definition of indistinguishability of machine states requires the program counters to be equal only if their labels are "low" compared to the observer label; otherwise they can be different. QuickChick quickly finds a counterexample if we use an indistinguishability relation that is too restrictive. During our proof efforts for the register machine, we initially got the "fix" wrong: we allowed one machine to be "high" while the other was "low". Such faulty definitions were not uncommon in our original designs; we used QuickChick to find much more subtle ones throughout our efforts. What makes this particular bug interesting however, is that our testing infrastructure (even MSNI) could not find

it at all!

The reason is, once again, that our generators for indistinguishable states were incomplete with respect to the (now faulty!) indistinguishability predicate. Indeed, our generator for indistinguishable machines never created starting configurations where one machine was in a "high" state and the other one in a "low" state, even though that was allowed by the indistinguishability relation. As a result, a large part of the state space was not exercised and a counterexample could not be produced. One of the main goals of this thesis is to tightly couple generators and predicates, so that such occurrences cease to exist.

3.3.2 Debugging Generators

In the extended machine, to avoid cluttering the step function with the IFC logic and injected bugs, we parameterized the step relation to take a *rule table* as an argument. A single rule would receive a number of labels as inputs (the *pc* label and the labels of any arguments), potentially perform checks and return labels for the result and the new *pc*. For example, the IFC entry for the *Store* instruction of the stack machine (whose semantics are repeated here for convenience)

$$\frac{i(pc) = \mathit{Store} \quad m(p) = n'@\ell'_n \quad \ell_p \vee \ell_{pc} \sqsubseteq \ell'_n \quad m' = m[p := n@(\ell_n \vee \ell_p \vee \ell_{pc})]}{\boxed{pc@\ell_{pc}} \; \boxed{p@\ell_p : n@\ell_n : s} \; \boxed{m} \Rightarrow \boxed{(pc{+}1)@\ell_{pc}} \; \boxed{s} \; \boxed{m'}} \quad \text{(STORE)}$$

would look like this:

Check	*Final pc Label*	*Result Label*
$\ell_p \vee \ell_{pc} \sqsubseteq \ell'_n$	ℓ_{pc}	$\ell_n \vee \ell_p \vee \ell_{pc}$

This factorization of IFC rules allowed a more systematic approach to *debugging* our generators. Since we are striving for a sound and permissive set of information-flow rules, every check performed and every tainting of result labels has to be *essential*; in other words, if we were to remove a check or a label join, our testing infrastructure should lead to a counterexample. By doing exactly that, we can systematically construct all possible *mutants* of a candidate rule in the rule table,

using the lattice structure of the labels. For example, the *Store* rule above gives rise to the following mutants, where each row depicts a single dropped taint or check.

Check	*Final pc Label*	*Result Label*
$\ell_{pc} \sqsubseteq \ell'_n$	ℓ_{pc}	$\ell_n \vee \ell_p \vee \ell_{pc}$
$\ell_p \sqsubseteq \ell'_n$	ℓ_{pc}	$\ell_n \vee \ell_p \vee \ell_{pc}$
True	ℓ_{pc}	$\ell_n \vee \ell_p \vee \ell_{pc}$
$\ell_p \vee \ell_{pc} \sqsubseteq \ell'_n$	$\bot$	$\ell_n \vee \ell_p \vee \ell_{pc}$
$\ell_p \vee \ell_{pc} \sqsubseteq \ell'_n$	ℓ_{pc}	$\ell_p \vee \ell_{pc}$
$\ell_p \vee \ell_{pc} \sqsubseteq \ell'_n$	ℓ_{pc}	$\ell_n \vee \ell_{pc}$
$\ell_p \vee \ell_{pc} \sqsubseteq \ell'_n$	ℓ_{pc}	$\ell_n \vee \ell_p$
$\ell_p \vee \ell_{pc} \sqsubseteq \ell'_n$	ℓ_{pc}	ℓ_n
$\ell_p \vee \ell_{pc} \sqsubseteq \ell'_n$	ℓ_{pc}	ℓ_p
$\ell_p \vee \ell_{pc} \sqsubseteq \ell'_n$	ℓ_{pc}	ℓ_{pc}
$\ell_p \vee \ell_{pc} \sqsubseteq \ell'_n$	ℓ_{pc}	$\bot$

If QuickChick cannot find a counterexample to a specific mutant, it has revealed something interesting: either the testing is not complete or the IFC rule is too strict! This is a particularly fortunate situation compared to standard mutation testing [71]: it completely avoids the "equivalent mutant problem" by construction. Unfortunately, preliminary attempts to generalize this approach to a more general setting like arbitrary inductive properties have failed. This is discussed further in future work (Chapter 8).

3.4 Shrinking

The counterexamples presented earlier on in this section are not the initial randomly generated machine states; they are the result of shrinking these to minimal counterexamples. For example, randomly generated counterexamples to EENI for the *Store* bugs usually consist of 20–40 instructions; the minimal counterexample we presented uses just 4.

Similarly to generation, one difficulty that arises when shrinking noninterference

counterexamples is that the test cases must be pairs of *indistinguishable* machines. Shrinking each machine state independently will most likely yield distinguishable pairs, which are invalid test cases, since they fail to satisfy the precondition of the property we are testing. In order to shrink effectively, we need to shrink both states of a variation *simultaneously*, and in the same way. For instance, if we shrink one machine state by deleting a *Noop* in the middle of its instruction memory, then we must delete the same instruction in the corresponding variation.

Initially, we used a shrink-and-test approach, were we shrunk variations of machine states and then discarded the ones that were not indistinguishable. To dramatically decrease the shrinking time, we needed to implement many heuristics, such as removing an instruction while decrementing some integer value at the same time to preserve relative jumps. Unfortunately, even though the generator combinator library of QuickChick is comprehensive and many common practices have been developed, support for sophisticated shrinking is somewhat lacking. Indeed, a general approach to "smart" shrinking in the presence of preconditions does not exist—yet. We also discuss this in the future work section 8.

3.5 Takeaways

In this chapter, we presented in some detail our work on testing noninterference, demonstrating that random testing can indeed be used to find bugs efficiently. When predicates with preconditions are involved, it is necessary to write custom generators for well-distributed random data satisfying those predicates. Some of the time, coming up with an efficient generator requires a lot of research effort and ingenuity (such as the generation by execution techniques). More often, even if writing a generator is a seemingly straightforward task (like varying a machine state to obtain an indistinguishable one), the process can be very error prone. Worse, the fact that generators and predicates are different artifacts that need to be kept in sync is a very real source of bugs. It would be much better to have a single program, preferably in the more declarative predicate form, and derive generators automatically. This is the main focus of the next two chapters and this dissertation in general.

We also identified two more open problems in our methodology that we leave for future work. The first is figuring out how much confidence you can really obtain from testing. Is the fact that no bugs can be found in a specification enough to make a proof go through? Currently, the only way to know for sure is to actually complete the proof! In the context of the register machine, we relied on a particularly useful formulation of mutation testing that revealed most flaws in our development. While work on mutation testing (and other ways of evaluating generators such as using coverage metrics) is both important and interesting, it is beyond the scope of this thesis.

The second is a more foundational treatment of shrinking. Shrinking tests while preserving some invariant is in a sense dual to generating random inputs satisfying some precondition. The naive method, shrink-and-test, where we filter potential shrinks based on the precondition, is as inefficient as generate-and-test is for producing random inputs. Coming up with more automated methods of writing specialized shrinkers that are tightly coupled with predicated, along with developing a *theory* of what it means for such shrinkers to be correct is an important task for the near future.

Acknowledgments

The experiments presented in this chapter were originally performed in Haskell using QuickCheck, in the context of the ICFP 2013 paper "Testing Noninterference, Quickly", with Catalin Hriţcu, John Hughes, Benjamin Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis and Arthur Azevedo de Amorim [65]. My main contribution in this line of work was extending the simple machine to the more complicated register machine and coming up with MSNI, leading to a journal version of the same paper [66]. Moreover, we've since ported all of these generators in QuickChick, to aid our efforts to prove the correctness of the design.

Chapter 4

Luck : A Language for Property-Based Generators

To enable effective property-based random testing of complex software artifacts, we need a better way of writing predicates and corresponding generators that are tightly coupled. A natural idea is to derive an efficient generator for a given predicate `p` directly from `p` itself. Indeed, two variants of this idea, with complementary strengths and weaknesses, have been explored by others—one based on local choices and backtracking, one on general constraint solving.

The first approach, which is often called *narrowing* in the literature, can be thought of as a kind of incremental generate-and-test: rather than generating completely random valuations and then testing them against `p`, we instead walk over the structure of `p` and instantiate each unknown variable `x` at the first point where we meet a constraint involving `x`. For example, consider the following standard `member` predicate:

```
member x (h:t) = (x == h) || member x t
member x [] = False
```

In `member`, following the narrowing approach, we make a random choice on each recursive call between the branches of the `||`. If we choose the left, we instantiate `x` to the head of the list; otherwise we leave `x` unknown and continue with the recursive call to `member` on the tail. Intuitively, just like a custom generator would,

we traverse the list and pick one of its elements. But we also need to be careful: naively picking each branch with probability 1/2 every time leads to a distribution which is heavily skewed towards the first elements.

We will refer to this local instantiation approach also as *narrowing*, because it resembles the narrowing mechanism in functional logic programming [1, 60, 87, 123]. It is attractively lightweight, admits natural control over distributions (as we will see in the next section), and has been used successfully [44, 30, 110, 48], even in challenging domains such as generating well-typed programs to test compilers [31, 43].

However, choosing a value for an unknown when we encounter the *first* constraint on it risks making choices that do not satisfy *later* constraints, forcing us to backtrack and make a different choice when the problem is discovered. For example, consider the `notMember` predicate:

```
notMember x (h:t)  =  (x /= h) && notMember x t
notMember x []     =  True
```

Suppose we wish to generate values for `x` such that `notMember x ys` for some predetermined list `ys`. When we first encounter the constraint `x /= h`, we generate a value for `x` that is not equal to the known value `h`. We then proceed to the recursive call of `notMember`, where we *check* that the chosen `x` does not appear in the rest of the list—we've essentially fallen back to the generate-and-test approach! Since the values in the rest of the list are not taken into account when choosing `x`, this may force us to backtrack if our choice of `x` was unlucky. If the space of possible values for `x` is not much bigger than the length of `ys`—say, just twice as big—then we will backtrack 50% of the time. Worse yet, if `notMember` is used to define another predicate—e.g., `distinct`, which tests whether each element of an input list is different from all the others—and we want to generate a list satisfying `distinct`, then `notMember`'s 50% chance of backtracking will be compounded on each recursive call of `distinct`, leading to unacceptably low rates of successful generation.

The second existing method that uses a predicate to obtain a generator leverages a *constraint solver* to generate a diverse set of valuations satisfying that pred-

icate.[1] This approach has been widely investigated, both for generating inputs directly from predicates [24, 116, 53, 79] and for symbolic-execution-based testing [49, 117, 20, 5, 124], which additionally uses the system under test to guide generation of inputs that exercise different control-flow paths. For `notMember`, gathering a set of disequality constraints on `x` before choosing its value avoids any backtracking.

However, *pure* constraint-solving approaches do not give us everything we need. They do not provide effective control over the distribution of generated valuations. At best, they might guarantee a *uniform* (or near-uniform) distribution [26], but this is typically not the distribution we want in practice (see §4.1). Moreover, the overhead of maintaining and solving constraints can make these approaches significantly less efficient than the more lightweight, local approach of needed narrowing when the latter does not lead to backtracking, as for instance in `member`.

The complementary strengths and weaknesses of local instantiation and global constraint solving suggest a hybrid approach, where limited constraint propagation, under explicit user control, is used to refine the domains (sets of possible values) of unknowns before instantiation. To explore this approach we designed Luck, a new domain-specific language for writing generators via lightweight annotations on predicates, combining the strengths of the local-instantiation and constraint-solving approaches to generation.

The main contributions of this chapter are organized as follows:

- We introduce this new domain-specific language, Luck, and illustrate its novel features using binary search trees as an example (Section 4.1).
- To place Luck's design on a firm formal foundation, we define a core calculus and establish key properties, including the soundness and completeness of its probabilistic generator semantics with respect to a straightforward interpretation of expressions as predicates (Sections 4.2 and 4.3).

[1] Constraint solvers can, of course, be used to *directly* search for counterexamples to a property of interest by software model checking [15, 68, 7, 70, etc.]. We are interested here in the rather different task of quickly generating a large number of diverse inputs, so that we can thoroughly test systems like compilers whose state spaces are too large to be exhaustively explored.

- We provide a prototype interpreter (Section 4.4) including a simple implementation of the constraint-solving primitives used by the generator semantics. We do not use an off-the shelf constraint solver because we want to experiment with a per-variable uniform sampling approach (as we will see in the next section) which is not supported by modern solvers. In addition, using such a solver would require translating Luck expressions—datatypes, pattern matching, etc.—into a form that it can handle. We leave this for future work.

- We evaluate Luck's expressiveness on a collection of common examples from the random testing literature (Section 4.5) and on two significant case studies; the latter demonstrate that Luck can be used (1) to find bugs in a widely used compiler (GHC) by randomly generating well-typed lambda terms and (2) to help design information-flow abstract machines by generating "low-indistinguishable" machine states, replicating the results of the case study of Chapter 3. Compared to hand-written generators, these experiments show comparable bug-finding effectiveness (measured in test cases generated per counterexample found) and a significant reduction in the size of testing code. The interpreted Luck generators run an order of magnitude slower than compiled QuickCheck versions (8 to 24 times per test), but many opportunities for optimization remain.

Auxiliary material for this chapter are available online: (1) a Coq formalization of the narrowing semantics of Luck and machine-checked proofs of its properties (available at `https://github.com/QuickChick/Luck`) (Section 4.2.3); (2) the prototype Luck interpreter and a battery of example programs, including all the ones we used for evaluation (also at `https://github.com/QuickChick/Luck`) (Section 4.5).

4.1 Luck by example

Figure 4.1 shows a recursive Haskell predicate `bst` that checks whether a given tree with labels strictly between `low` and `high` satisfies the standard binary-search tree

(BST) invariant [97]. It is followed by a QuickCheck generator `genTree`, which generates BSTs with a given maximum depth, controlled by the `size` parameter. This generator first checks whether `low + 1 >= high`, in which case it returns the only valid BST satisfying this constraint—the `Empty` one. Otherwise, it uses QuickCheck's `frequency` combinator, which takes a list of pairs of positive integer weights and associated generators and randomly selects one of the generators using the probabilities specified by the weights. In this example, $\frac{1}{size+1}$ of the time it creates an `Empty` tree, while $\frac{size}{size+1}$ of the time it returns a `Node`. The `Node` generator is specified using monadic syntax: first it generates an integer `x` that is strictly between `low` and `high`, and then the left and right subtrees `l` and `r` by calling `genTree` recursively; finally it returns `Node x l r`.

The generator for BSTs allows us to efficiently test conditional properties of the form "if `bst t` then ⟨*some other property of* `t`⟩," but it raises some new issues of its own. First, even for this simple example, getting the generator right is a bit tricky (for instance because of potential off-by-one errors in generating `x`), and it is not immediately obvious that the set of trees generated by the generator is exactly the set accepted by the predicate. Worse, we now need to maintain two similar but distinct artifacts and keep them in sync. (We can't just throw away the predicate and keep the generator because we often need them both, for example to test properties like "the `insert` function applied to a BST and a value returns a BST.") As predicates and generators become more complex, these issues can become quite problematic (e.g., [65]).

Enter Luck. The bottom of Figure 4.1 shows a Luck program that represents *both* a BST predicate *and* a generator for random BSTs. Modulo variations in concrete syntax, the Luck code follows the Haskell `bst` predicate quite closely. The significant differences are: (1) the *sample-after expression* `!x`, which controls when node labels are generated, and (2) the `size` parameter, which is used, as in the QuickCheck generator, to annotate the branches of the `case` with relative weights. Together, these enable us to give the program both a natural interpretation as a predicate (by simply ignoring weights and sampling expressions) and an efficient interpretation as a generator of random trees with the same distribution as the QuickCheck version. For example, evaluating the top-level query `bst 10 0 42`

Binary tree datatype (in both Haskell and Luck):

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Test predicate for BSTs (in Haskell):

```
bst :: Int -> Int -> Tree Int -> Bool
bst low high tree =
  case tree of
    Empty -> True
    Node x l r ->
      low < x && x < high
      && bst low x l && bst x high r
```

QuickCheck generator for BSTs (in Haskell):

```
genTree :: Int -> Int -> Int -> Gen (Tree Int)
genTree size low high
  | low + 1 >= high = return Empty
  | otherwise =
      frequency [(1, return Empty),
                 (size, do
                   x <- choose (low + 1, high - 1)
                   l <- genTree (size `div` 2) low x
                   r <- genTree (size `div` 2) x high
                   return (Node x l r))]
```

Luck generator (and predicate) for BSTs:

```
sig bst :: Int -> Int -> Int -> Tree Int -> Bool
fun bst size low high tree =
  if size == 0 then tree == Empty
  else case tree of
       | 1    % Empty -> True
       | size % Node x l r ->
          ((low < x && x < high) !x)
          && bst (size / 2) low x l
          && bst (size / 2) x high r
```

Figure 4.1: Binary Search Tree Checker and Two Generators

`u = True`—i.e., "generate values `t` for the unknown `u` such that `bst 10 0 42 t` evaluates to `True`"—will yield random binary search trees of size up to 10 with node labels strictly between 0 and 42, with the same distribution as the QuickCheck generator `genTree 10 0 42`.

An *unknown* in Luck is a special kind of value, similar to logic variables found in logic programming languages and unification variables used by type-inference algorithms. Unknowns are typed, and each is associated with a domain of possible values from its type. Given an expression e mentioning some set U of unknowns, our goal is to generate *valuations* over these unknowns (maps from U to concrete values) by iteratively refining the unknowns' domains, so that, when any of these valuations is substituted into e, the resulting concrete term evaluates to a desired value (e.g., `True`).

Unknowns can be introduced both explicitly, as in the top-level query above, and implicitly, as in the generator semantics of `case` expressions. In the `bst` example, when the `Node` branch is chosen, the pattern variables `x`, `l`, and `r` are replaced by fresh unknowns, which are then instantiated by evaluating the constraint `low < x && x < high` and the recursive calls to `bst`.

Varying the placement of unknowns in the top-level `bst` query yields different behaviors. For instance, if we change the query to `bst 10 ul uh u = True`, replacing the `low` and `high` parameters with unknowns `ul` and `uh`, the domains of these unknowns will be refined during tree generation and the result will be a generator for random valuations (`ul` $\mapsto$ `i`, `uh` $\mapsto$ `j`, `u` $\mapsto$ `t`) where `i` and `j` are lower and upper bounds on the node labels in `t`.

Alternatively, we can evaluate the top-level query `bst 10 0 42 t = True`, replacing `u` with a concrete tree `t`. In this case, Luck will return a trivial valuation only if `t` is a binary search tree; otherwise it will report that the query is unsatisfiable. A less useful possibility is that we provide explicit values for `low` and `high` but choose them with `low` $>$ `high`, e.g., `bst 10 6 4 u = True`. Since there are no satisfying valuations for `u` other than `Empty`, Luck will now generate only `Empty` trees.

A *sample-after expression* of the form `e !x` is used to control instantiation of unknowns. Typically, `x` will be an unknown `u`, and evaluating `e !u` will cause `u`

to be instantiated to a concrete value (after evaluating `e` to refine the domains of all of the unknowns in `e`). If `x` reduces to a value rather than an unknown, we similarly instantiate any unknowns appearing within this value.

As a concrete example, consider the compound inequality constraint `0 < x && x < 4`. A generator based on pure narrowing (as in [48]), would instantiate `x` when the evaluator meets the first constraint where it appears, namely `0 < x` (assuming left-to-right evaluation order). We can mimic this behavior in Luck by writing `((0 < x) !x) && (x < 4)`. However, picking a value for `x` at this point ignores the constraint `x < 4`, which can lead to backtracking. If, for instance, the domain from which we are choosing values for `x` is 32-bit integers, then the probability that a random choice satisfying `0 < x` will also satisfy `x < 4` is minuscule. It is better in this case to write `(0 < x && x < 4) !x`, instantiating `x` after the entire conjunction has been evaluated and all the constraints on the domain of `x` recorded and thus avoiding backtracking completely. Finally, if we do not include a sample-after expression for `x` here at all, we can further refine its domain with constraints later on, at the cost of dealing with a more abstract representation of it internally in the meantime. Thus, sample-after expressions give Luck users explicit control over the tradeoff between the expense of possible backtracking—when unknowns are instantiated early—and the expense of maintaining constraints on unknowns—so that they can be instantiated late (e.g., so that `x` can be instantiated after the recursive calls to `bst`).

Sample-after expressions choose random values with *uniform* probability from the domain associated with each unknown. While this behavior is sometimes useful, effective property-based random testing often requires fine control over the distribution of generated test cases. Drawing inspiration from the QuickCheck combinator library for building complex generators, and particularly `frequency`, Luck also allows weight annotations on the branches of a `case` expression which have a `frequency`-like effect. In the Luck version of `bst`, for example, the unknown `tree` is either instantiated to an `Empty` tree $\frac{1}{1+size}$ of the time or partially instantiated to a `Node` (with fresh unknowns for `x` and the left and right subtrees) $\frac{size}{1+size}$ of the time.

Weight annotations give the user control over the probabilities of local choices.

These do not necessarily correspond to a specific posterior probability, but the random testing community has established techniques for guiding the user in tuning local weights to obtain good testing. For example, we the user can wrap properties inside a `collect x` combinator that we also encountered in the QuickChick tutorial (Chapter 2); during testing, QuickCheck will gather information on `x`, grouping equal values to provide an estimate of the posterior distribution that is being sampled. The `collect` combinator is an effective tool for adjusting `frequency` weights and dramatically increasing bug-finding rates (e.g., [65]). The Luck implementation provides a similar primitive.

One further remark on uniform sampling: while *locally* instantiating unknowns uniformly from their domain is a useful default, generating *globally* uniform distributions of test cases is usually not what we want, as this often leads to inefficient testing in practice. A simple example can be drawn from the information-flow-control experiments of the previous Chapter. Consider the indistinguishability predicate in Haskell ("high" elements are always indistinguishable, "low" ones require equal payloads):

```
indist (v1,H) (v2,H)  =  True
indist (v1,L) (v2,L)  =  v1 == v2
indist _      _       =  False
```

If we use 32-bit integers, then for every `Low` indistinguishable pair there are 2^{32} `High` ones! Thus, choosing a uniform distribution over indistinguishable pairs means that we will essentially never generate pairs with `Low` labels. Clearly, such a distribution cannot provide effective testing; indeed, in our experiments we discovered that the best distribution was actually somewhat skewed in favor of `Low` labels.

4.2 Semantics of Core Luck

We next present a core calculus for Luck—a minimal subset into which the examples in the previous section can in principle be desugared (though our implementation does not do this). The core omits primitive booleans and integers and replaces datatypes with binary sums, products, and iso-recursive types.

We begin in Figure 4.2.1 with the syntax and standard *predicate semantics* of the core. We call it the "predicate" semantics because, in our examples, the result of evaluating a top-level expression will typically be a boolean, though this expectation is not baked into the formalism. We then build up to the full generator semantics in three steps. First, we give an interface to a *constraint solver* (Section 4.2.2), abstracting over the primitives required to implement our semantics. Then we define a probabilistic *narrowing semantics*, which enhances the local-instantiation approach to random generation with QuickCheck-style distribution control (Section 4.2.3). Finally, we introduce a *matching semantics*, building on the narrowing semantics, that unifies constraint solving and narrowing into a single evaluator (Section 4.2.4). We also show how integers and booleans can be encoded and how the semantics applies to the binary search tree example (Section 4.2.5) The key properties of the generator semantics (both narrowing and matching versions) are soundness and completeness with respect to the predicate semantics; we present them in the following section (Section 4.3). Informally, whenever we use a Luck program to generate a valuation that satisfies some predicate, the valuation will satisfy the boolean predicate semantics (soundness), and it will generate every possible satisfying valuation with non-zero probability (completeness).

4.2.1 Syntax, Typing, and Predicate Semantics

The syntax of Core Luck is given in Figure 4.2. Except for the last line in the definitions of values and expressions, it is a standard simply typed call-by-value lambda calculus with sums, products, and iso-recursive types. We include recursive lambdas for convenience in examples, although in principle they could be encoded using recursive types.

Values include unit, pairs of values, sum constructors (L and R) applied to values (and annotated with types, to eliminate ambiguity), first class (potentially) recursive functions (*rec*), *fold*-annotated values indicating where an iso-recursive type should be "folded," and *unknowns* drawn from an infinite set. The standard expression forms include variables, unit, functions, function applications, pairs with a single-branch pattern-matching construct for deconstructing them, value tagging

$$
\begin{aligned}
v &::= () \mid (v, v) \mid L_T\ v \mid R_T\ v \\
&\quad \mid \mathit{rec}\ (f : T_1 \to T_2)\ x = e \mid \mathit{fold}_T\ v \\
&\quad \mid u \\
e &::= x \mid () \mid \mathit{rec}\ (f : T_1 \to T_2)\ x = e \mid (e\ e) \\
&\quad \mid (e, e) \mid \mathit{case}\ e\ \mathit{of}\ (x, y) \to e \\
&\quad \mid L_T\ e \mid R_T\ e \mid \mathit{case}\ e\ \mathit{of}\ (L\ x \to e)\ (R\ x \to e) \\
&\quad \mid \mathit{fold}_T\ e \mid \mathit{unfold}_T\ e \\
&\quad \mid u \mid e \leftarrow (e, e) \mid\ !e \mid e\ ;\ e \\
\overline{T} &::= X \mid 1 \mid \overline{T} + \overline{T} \mid \overline{T} \times \overline{T} \mid \mu X.\ \overline{T} \\
T &::= X \mid 1 \mid T + T \mid T \times T \mid \mu X.\ T \mid T \to T \\
\Gamma &::= \varnothing \mid \Gamma, x : T
\end{aligned}
$$

Figure 4.2: Core Luck Syntax

(L and R), pattern matching on tagged values, and *fold*/*unfold*. The nonstandard additions are unknowns (u), *instantiation* ($e \leftarrow (e_1, e_2)$), *sample* ($!e$) and *after* ($e_1\ ;\ e_2$) expressions.

The "after" operator, written with a backwards semicolon, evaluates both e_1 and e_2 in sequence. However, unlike the standard sequencing operator $e_1; e_2$, the result of $e_1\ ;\ e_2$ is the result of e_1; the expression e_2 is evaluated just for its side-effects. For example, the sample-after expression `e !x` of the previous section is desugared to a combination of sample and after: $e\ ;\ !x$. If we evaluate this snippet in a context where x is bound to some unknown u, then the expression e is evaluated first, refining the domain of u (amongst other unknowns); then the sample expression $!u$ is evaluated for its side effect, instantiating u to a uniformly generated value from its domain; and finally the result of e is returned as the result of the whole expression. A reasonable way to implement $e_1\ ;\ e_2$ using standard lambda abstractions would be as $(\lambda\ x.\ (\lambda\_.\ x)\ e_2)\ e_1$. However, there is a slight difference in the semantics of this encoding compared to our intended semantics—we will return to this point in Section 4.2.4.

Weight annotations like the ones in the `bst` example can be desugared using

instantiation expressions. For example, assuming a standard encoding of binary search trees ($\mathit{Tree} = \mu X.\ 1 + \mathit{int} \times X \times X$) and naturals, plus syntactic sugar for constant naturals:

$$\mathit{case}\ (\mathit{unfold}_{\mathit{Tree}}\ \mathit{tree} \leftarrow (1, \mathit{size}))\ \mathit{of}\ (L\ x \rightarrow \ldots)(R\ y \rightarrow \ldots)$$

Most of the typing rules are standard (these can be found in Fig. 4.3). The four non-standard rules are given in Fig. 4.4. Unknowns are typed: each will be associated with a domain (set of values) drawn from a type $\overline{T}$ that does not contain arrows. Luck does not support constraint solving over functional domains (which would require something like higher-order unification), and the restriction of unknowns to non-functional types reflects this. To remember the types of unknowns, we extend the typing context to include a component U, a map from unknowns to non-functional types. When the variable typing environment $\Gamma = \varnothing$, we write $U \vdash e : T$ as a shorthand for $\varnothing; U \vdash e : T$. The rules for the standard constructs in Fig. 4.3 are as expected (adding U everywhere). An unknown u has type $\overline{T}$ if $U(u) = \overline{T}$. If e_1 and e_2 are well typed, then $e_1\ ;\ e_2$ shares the type of e_1. An instantiation expression $e \leftarrow (e_l, e_r)$ is well typed if e has sum type $\overline{T}_1 + \overline{T}_2$ and e_l and e_r are natural numbers. A sample expression $!e$ has the (non-functional) type $\overline{T}$ when e has type $\overline{T}$.

The predicate semantics for Core Luck, written $e \Downarrow v$, are defined as a big-step operational semantics. We assume that e is closed with respect to ordinary variables and free of unknowns. The rules for the standard constructs are unsurprising (Figure 4.5). The only non-standard rules are the ones for narrow, sample and after expressions, which are essentially ignored (Figure 4.6). With the predicate semantics we can implement a naive generate-and-test method for generating valuations satisfying some predicate by generating arbitrary well-typed valuations and filtering out those for which the predicate does not evaluate to `True`.

4.2.2 Constraint Sets

The rest of this section develops an alternative probabilistic generator semantics for Core Luck. This semantics will use *constraint sets* (whose type we denote as

$$\textbf{T-Var}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \textbf{T-Unit}\ \frac{}{\Gamma \vdash () : 1}$$

$$\textbf{T-Abs}\ \frac{\Gamma, x : T_1, f : T_1 \to T_2 \vdash e_2 : T_2}{\Gamma \vdash \mathit{rec}\ (f : T_1 \to T_2)\ x = e_2 : T_1 \to T_2}$$

$$\textbf{T-App}\ \frac{\Gamma \vdash e_0 : T_1 \to T_2 \qquad \Gamma \vdash e_1 : T_1}{\Gamma \vdash (e_0\ e_1) : T_2}$$

$$\textbf{T-Pair}\ \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : (T_1 \times T_2)}$$

$$\textbf{T-CasePair}\ \frac{\begin{array}{c}\Gamma \vdash e : (T_1 \times T_2) \\ \Gamma, x : T_1, y : T_2 \vdash e' : T\end{array}}{\Gamma \vdash \mathit{case}\ e\ \mathit{of}\ (x, y) \to e' : T}$$

$$\textbf{T-L}\ \frac{\Gamma \vdash e : T_1}{\Gamma \vdash L_{T_1+T_2}\ e : T_1 + T_2}$$

$$\textbf{T-R}\ \frac{\Gamma \vdash e : T_2}{\Gamma \vdash R_{T_1+T_2}\ e : T_1 + T_2}$$

$$\textbf{T-Case}\ \frac{\begin{array}{c}\Gamma \vdash e : T_1 + T_2 \\ \Gamma, x : T_1 \vdash e_1 : T \qquad \Gamma, y : T_2 \vdash e_2 : T\end{array}}{\Gamma \vdash \mathit{case}\ e\ \mathit{of}\ (\mathit{inl}\ x \to e_1)\ (\mathit{inr}\ y \to e_2) : T}$$

$$\textbf{T-Fold}\ \frac{U = \mu X.\ T_1 \qquad \Gamma \vdash e_1 : T_1[U/X]}{\Gamma \vdash \mathit{fold}_U\ e_1 : U}$$

$$\textbf{T-Unfold}\ \frac{U = \mu X.\ T_1 \qquad \Gamma \vdash e_1 : U}{\Gamma \vdash \mathit{unfold}_U\ e_1 : T_1[U/X]}$$

Figure 4.3: Standard Typing Rules

$$\textbf{T-U}\ \frac{U(u) = \overline{T}}{\Gamma; U \vdash u : \overline{T}} \qquad \textbf{T-After}\ \frac{\Gamma; U \vdash e_1 : T_1 \qquad \Gamma; U \vdash e_2 : T_2}{\Gamma; U \vdash e_1 \,;\, e_2 : T_1}$$

$$\textbf{T-Bang}\ \frac{\Gamma; U \vdash e : \overline{T}}{\Gamma; U \vdash !e : \overline{T}} \qquad \textbf{T-Narrow}\ \frac{\Gamma; U \vdash e : \overline{T}_1 + \overline{T}_2 \qquad \Gamma; U \vdash e_l : \mathit{nat} \qquad \Gamma \vdash e_r : \mathit{nat}}{\Gamma; U \vdash e \leftarrow (e_l, e_r) : \overline{T}_1 + \overline{T}_2}$$

$$\mathit{nat} := \mu X.\ 1 + X$$

Figure 4.4: Typing Rules for Nonstandard Constructs

$\mathcal{C}$) to describe the possible values that unknowns can take. For the moment, we leave the implementation of constraint sets open (the one used by our prototype interpreter is described in Section 4.4), simply requiring that they support the following operations:

$$\begin{array}{lcl}
[\![\cdot]\!] & :: & \mathcal{C} \to \mathit{Set}\ \ \mathit{Valuation} \\
U & :: & \mathcal{C} \to \mathit{Map}\ \mathcal{U}\ \overline{T} \\
\mathit{fresh} & :: & \mathcal{C} \to \overline{T}^* \to (\mathcal{C} \times \mathcal{U}^*) \\
\mathit{unify} & :: & \mathcal{C} \to \mathit{Val} \to \mathit{Val} \to \mathcal{C} \\
\mathit{SAT} & :: & \mathcal{C} \to \mathit{Bool} \\
[\cdot] & :: & \mathcal{C} \to \mathcal{U} \to \mathit{Maybe}\ \ \mathit{Val} \\
\mathit{sample} & :: & \mathcal{C} \to \mathcal{U} \to \mathcal{C}^*
\end{array}$$

Here we describe these operations informally, deferring technicalities until after we have presented the generator semantics (Section 4.3).

A constraint set κ denotes a set of valuations (mappings from variables to values, denoted $[\![\kappa]\!]$), representing the solutions to the constraints. Constraint sets also carry type information about existing unknowns: $U(\kappa)$ is a mapping from κ's unknowns to types. A constraint set κ is *well typed* ($\vdash \kappa$) if, for every valuation σ in the denotation of κ and every unknown u bound in σ, the type map $U(\kappa)$

$$\textbf{P-Val}\ \frac{\mathit{is\_value}\ v}{v \Downarrow v}$$

$$\textbf{P-App}\ \frac{\begin{array}{c} e_0 \Downarrow (\mathit{rec}\ (f : T_1 \to T_2)\ x = e_2) \\ e_1 \Downarrow v_1 \\ e[(\mathit{rec}\ (f : T_1 \to T_2)\ x = e_2)/f, v_1/x] \Downarrow v \end{array}}{(e_0\ e_1) \Downarrow v}$$

$$\textbf{P-Pair}\ \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(e_1, e_2) \Downarrow (v_1, v_2)}$$

$$\textbf{P-CasePair}\ \frac{\begin{array}{c} e \Downarrow (v_1, v_2) \\ e'[v_1/x, v_2/y] \Downarrow v \end{array}}{\mathit{case}\ e\ \mathit{of} (x, y) \to e' \Downarrow v}$$

$$\textbf{P-L}\ \frac{e \Downarrow v}{L_T\ e \Downarrow L_T\ v}$$

$$\textbf{P-R}\ \frac{e \Downarrow v}{R_T\ e \Downarrow R_T\ v}$$

$$\textbf{P-Case-L}\ \frac{e \Downarrow L_T\ v \qquad e_1[v/x] \Downarrow v_1}{\mathit{case}\ e\ \mathit{of}\ (L\ x \to e_1)(R\ y \to e_2) \Downarrow v_1}$$

$$\textbf{P-Case-R}\ \frac{e \Downarrow R_T\ v \qquad e_2[v/y] \Downarrow v_2}{\mathit{case}\ e\ \mathit{of}\ (L\ x \to e_1)(R\ y \to e_2) \Downarrow v_2}$$

$$\textbf{P-Fold}\ \frac{e \Downarrow v}{\mathit{fold}_S\ e \Downarrow \mathit{fold}_S\ v}$$

$$\textbf{P-Unfold}\ \frac{e \Downarrow \mathit{fold}_T\ v}{\mathit{unfold}_T\ e \Downarrow v}$$

Figure 4.5: Predicate Semantics for Standard Core Luck Constructs

$$\textbf{P-Narrow}\ \frac{e \Downarrow v \quad e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad [\![v_1]\!] > 0 \quad [\![v_2]\!] > 0}{e \leftarrow (e_1, e_2) \Downarrow v} \qquad \textbf{P-Bang}\ \frac{e \Downarrow v}{!e \Downarrow v}$$

$$\textbf{P-After}\ \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \,;\, e_2 \Downarrow v_1}$$

$$\begin{array}{rcl} [\![\mathit{fold}_{nat}\ (L_{1+nat}\ ())]\!] & = & 0 \\ [\![\mathit{fold}_{nat}\ (R_{1+nat}\ v)]\!] & = & 1 + [\![v]\!] \end{array}$$

Figure 4.6: Predicate Semantics for Nonstandard Constructs

contains u and the binding of u in σ has the type prescribed by $U(\kappa)$:

$$\forall(\sigma \in [\![\kappa]\!])(u \in \sigma).\ \ u \in U(\kappa) \wedge \varnothing; U(\kappa) \vdash \sigma(u) : U(\kappa)(u)$$

Many of the semantic rules will need to introduce fresh unknowns. The *fresh* function takes as inputs a constraint set κ and a sequence of (non-functional) types of length k; it draws the next k unknowns (in some deterministic order) from the infinite set $\mathcal{U}$ and extends $U(\kappa)$ with the respective bindings.

The main way constraints are introduced during evaluation is unification. Given a constraint set κ and two values, each potentially containing unknowns, *unify* updates κ to preserve only those valuations in which the values match.

SAT is a total predicate that holds on constraint sets whose denotation contains at least one valuation. The totality requirement implies that our constraints must be decidable.

The value-extraction function $\kappa[u]$ returns an optional (non-unknown) value: if in the denotation of κ, all valuations map u to the same value v, then that value is returned (written $\{v\}$); otherwise nothing (written $\varnothing$).

The *sample* operation is used to implement sample expressions ($!e$): given a constraint set κ and an unknown $u \in U(\kappa)$, it returns a list of constraint sets representing all possible concrete choices for u, in all of which u is completely determined—that is $\forall \kappa \in (\mathit{sample}\ \kappa\ u).\ \exists v.\ \kappa[u] = \{v\}$. To allow for reasonable

implementations of this interface, we maintain an invariant that the input unknown to *sample* will always have a finite denotation; thus, the resulting list is also finite.

4.2.3 Narrowing Semantics

As a first step toward a semantics for Core Luck that incorporates both constraint solving and local instantiation, we define a simpler *narrowing* semantics. This semantics is of some interest in its own right, in that it extends traditional "needed narrowing" with explicit probabilistic instantiation points, but its role here is as a subroutine of the matching semantics in Section 4.2.4.

The narrowing evaluation judgment takes as inputs an expression e and a constraint set κ. As in the predicate semantics, evaluating e returns a value v, but now it also depends on a constraint set κ and returns a new constraint set κ'. The latter is intuitively a refinement of κ—i.e., evaluation will only remove valuations.

$$e \rightleftharpoons \kappa \Downarrow_q^t \kappa' \vDash v$$

The semantics is annotated with a representation of the sequence of random choices made during evaluation, in the form of a *trace* t. A trace is a sequence of *choices*: integer pairs (m, n) with $0 \leqslant m < n$, where n denotes the number of possibilities chosen among and m is the index of the one actually taken. We write ϵ for the empty trace and $t \cdot t'$ for the concatenation of two traces. We also annotate the judgment with the probability q of making the choices represented in the trace. Recording traces is useful after the fact in calculating the total probability of some given outcome of evaluation (which may be reached by many different derivations). Traces play no role in determining how evaluation proceeds. We model probability distributions using rational numbers $q \in (0, 1] \cap \mathbb{Q}$, for simplicity in the Coq formalization.

We maintain the invariant that the input constraint set κ is well typed and that the input expression e is well typed with respect to an empty variable context and the unknown context $U(\kappa)$. Another invariant is that every constraint set κ that appears as input to a judgment is satisfiable and the restriction of its denotation to the unknowns in e is finite. These invariants are established at the top-level (see

Section 4.4.1). The finiteness invariant ensures the output of *sample* will always be a finite collection (and therefore the probabilities involved will be positive rational numbers. Moreover, they guarantee termination of constraint solving, as we will see in Section 4.2.4. Finally, we assume that the type of every expression has been determined by an initial type-checking phase. We write e^T to show that e has type T. This information is used in the semantic rules to provide types for fresh unknowns.

The narrowing semantics is given in Figures 4.7 and 4.8 for the standard constructs and in Figure 4.9 for instantiation expressions; Figures 4.10 and 4.11 give some auxiliary definitions. Most of the rules are intuitive. A common pattern is sequencing two narrowing judgments $e_1 \rightleftharpoons \kappa \Downarrow_{q_1}^{t_1} \kappa_1 \vDash v$ and $e_2 \rightleftharpoons \kappa_1 \Downarrow_{q_2}^{t_2} \kappa_2 \vDash v$. The constraint-set result of the first narrowing judgment (κ_1) is given as input to the second, while traces and probabilities are accumulated by concatenation ($t_1 \cdot t_2$) and multiplication ($q_1 * q_2$). We now explain the rules in detail.

Rule **N-Base** is the base case of the evaluation relation, handling values that are not handled by other rules by returning them as-is. No choices are made, so the probability of the result is 1 and the trace is empty.

Rule **N-Pair**: To evaluate (e_1, e_2) given a constraint set κ, we sequence the derivations for e_1 and e_2.

Rules **N-CasePair-P**, **N-CasePair-U**: To evaluate the pair elimination expression *case* e *of* $(x, y) \rightarrow e'$ in a constraint set κ, we first evaluate e in κ. Typing ensures that the resulting value is either a pair or an unknown. If it is a pair (**N-CasePair-P**), we substitute its components for x and y in e' and continue evaluating. If it is an unknown u of type $\overline{T}_1 \times \overline{T}_2$ (**N-CasePair-U**), we first use $\overline{T}_1$ and $\overline{T}_2$ as types for fresh unknowns u_1, u_2 and remember the constraint that the pair (u_1, u_2) must unify with u. We then proceed as above, this time substituting u_1 and u_2 for x and y.

(The first pair rule might appear unnecessary since, even in the case where the scrutinee evaluates to a pair, we could generate unknowns, unify, and substitute, as in **N-CasePair-U**. However, unknowns in Luck only range over non-functional types $\overline{T}$, so this trick does not work when the type of the e contains arrows.)

The **N-CasePair-U** rule also shows how the finiteness invariant is preserved:

$$\textbf{N-Base}\frac{v = () \ \vee \ v = (rec\ (f : T_1 \to T_2)\ x = e') \ \vee \ v \in \mathcal{U}}{v \dashv \kappa \Downarrow_1^{\epsilon} \kappa \vDash v}$$

$$\textbf{N-Pair}\frac{e_1 \dashv \kappa \Downarrow_{q_1}^{t_1} \kappa_1 \vDash v_1 \qquad e_2 \dashv \kappa_1 \Downarrow_{q_2}^{t_2} \kappa_2 \vDash v_2}{(e_1, e_2) \dashv \kappa \Downarrow_{q_1 * q_2}^{t_1 \cdot t_2} \kappa_2 \vDash (v_1, v_2)}$$

$$\textbf{N-CasePair-P}\ \frac{\begin{array}{c} e \dashv \kappa \Downarrow_q^t \kappa_a \vDash (v_1, v_2) \\ e'[v_1/x,\ v_2/y] \dashv \kappa_a \Downarrow_{q'}^{t'} \kappa' \vDash v \end{array}}{case\ e\ of\ (x, y) \to e' \dashv \kappa \Downarrow_{q*q'}^{t \cdot t'} \kappa' \vDash v}$$

$$\textbf{N-CasePair-U}\ \frac{\begin{array}{c} e \dashv \kappa \Downarrow_q^t \kappa_a \vDash u \\ (\kappa_b, [u_1, u_2]) = fresh\ \kappa_a\ [\overline{T}_1, \overline{T}_2] \\ \kappa_c = unify\ \kappa_b\ (u_1, u_2)\ u \\ e'[u_1/x,\ u_2/y] \dashv \kappa_c \Downarrow_{q'}^{t'} \kappa' \vDash v \end{array}}{case\ e^{\overline{T}_1 \times \overline{T}_2}\ of\ (x, y) \to e' \dashv \kappa \Downarrow_{q*q'}^{t \cdot t'} \kappa' \vDash v}$$

$$\textbf{N-L}\ \frac{e \dashv \kappa \Downarrow_q^t \kappa' \vDash v}{L_{T_1+T_2}\ e \dashv \kappa \Downarrow_q^t \kappa' \vDash L_{T_1+T_2}\ v} \qquad \textbf{N-R}\ \frac{e \dashv \kappa \Downarrow_q^t \kappa' \vDash v}{R_{T_1+T_2}\ e \dashv \kappa \Downarrow_q^t \kappa' \vDash R_{T_1+T_2}\ v}$$

$$\textbf{N-Case-L}\ \frac{\begin{array}{c} e \dashv \kappa \Downarrow_q^t \kappa_a \vDash L_T\ v_l \\ e_l[v_l/x_l] \dashv \kappa_a \Downarrow_{q'}^{t'} \kappa' \vDash v \end{array}}{case\ e\ of\ (L\ x_l \to e_l)(R\ x_r \to e_r) \dashv \kappa \Downarrow_{q*q'}^{t \cdot t'} \kappa' \vDash v}$$

$$\textbf{N-Case-R}\ \frac{\begin{array}{c} e \dashv \kappa \Downarrow_q^t \kappa_a \vDash R_T\ v_r \\ e_r[v_r/x_r] \dashv \kappa_a \Downarrow_{q'}^{t'} \kappa' \vDash v \end{array}}{case\ e\ of\ (L\ x_l \to e_l)(R\ x_r \to e_r) \dashv \kappa \Downarrow_{q*q'}^{t \cdot t'} \kappa' \vDash v}$$

$$\textbf{N-Case-U}\ \frac{\begin{array}{c} e \dashv \kappa \Downarrow_{q_1}^{t_1} \kappa_a \vDash u \\ (\kappa_0, [u_l, u_r]) = fresh\ \kappa_a\ [\overline{T}_l, \overline{T}_r] \\ \kappa_l = unify\ \kappa_0\ u\ (L_{\overline{T}_l+\overline{T}_r}\ u_l) \qquad \kappa_r = unify\ \kappa_0\ u\ (R_{\overline{T}_l+\overline{T}_r}\ u_r) \\ choose\ 1\ \kappa_l\ 1\ \kappa_r \to_{q_2}^{t_2} i \\ e_i[u_i/x_i] \dashv \kappa_i \Downarrow_{q_3}^{t_3} \kappa' \vDash v \end{array}}{case\ e^{\overline{T}_l+\overline{T}_r}\ of\ (L\ x_l \to e_l)(R\ x_r \to e_r) \dashv \kappa \Downarrow_{q_1*q_2*q_3}^{t_1 \cdot t_2 \cdot t_3} \kappa' \vDash v}$$

Figure 4.7: Narrowing Semantics of Standard Core Luck Constructs (part 1)

$$\textbf{N-App}\ \frac{\begin{array}{c} e_0 \dashv \kappa \Downarrow_{q_0}^{t_0} \kappa_a \vDash (\mathit{rec}\ (f : T_1 \to T_2)\ x = e_2) \\ e_1 \dashv \kappa_a \Downarrow_{q_1}^{t_1} \kappa_b \vDash v_1 \\ e_2[(\mathit{rec}\ (f : T_1 \to T_2)\ x = e_2)/f,\ v_1/x] \dashv \kappa_b \Downarrow_{q_2}^{t_2} \kappa' \vDash v \end{array}}{(e_0\ e_1) \dashv \kappa \Downarrow_{q_0 * q_1 * q_2}^{t_0 \cdot t_1 \cdot t_2} \kappa' \vDash v}$$

$$\textbf{N-Fold}\frac{e \dashv \kappa \Downarrow_q^t \kappa' \vDash v}{\mathit{fold}_T\ e \dashv \kappa \Downarrow_q^t \kappa' \vDash \mathit{fold}_T\ v}$$

$$\textbf{N-Unfold-F}\ \frac{e \dashv \kappa \Downarrow_q^t \kappa' \vDash \mathit{fold}_T\ v}{\mathit{unfold}_T\ e \dashv \kappa \Downarrow_q^t \kappa' \vDash v}$$

$$\textbf{N-Unfold-U}\frac{\begin{array}{c} e \dashv \kappa \Downarrow_q^t \kappa_a \vDash u \\ (\kappa_b, u') = \mathit{fresh}\ \kappa_a\ T[\mu X.T/X] \\ \kappa' = \mathit{unify}\ \kappa_b\ u\ (\mathit{fold}_{\mu X.T}\ u') \end{array}}{\mathit{unfold}_{\mu X.T}\ e \dashv \kappa \Downarrow_q^t \kappa' \vDash u'}$$

Figure 4.8: Narrowing Semantics of Standard Core Luck Constructs (part 2)

$$\textbf{N-After}\frac{e_1 \dashv \kappa \Downarrow_{q_1}^{t_1} \kappa_1 \vDash v_1 \qquad e_2 \dashv \kappa_1 \Downarrow_{q_2}^{t_2} \kappa_2 \vDash v_2}{e_1\ ;\ e_2 \dashv \kappa \Downarrow_{q_1 * q_2}^{t_1 \cdot t_2} \kappa_2 \vDash v_1}$$

$$\textbf{N-Bang}\frac{e \dashv \kappa \Downarrow_q^t \kappa_a \vDash v \qquad \mathit{sampleV}\ \kappa_a\ v \Rightarrow_{q'}^{t'} \kappa'}{!e \dashv \kappa \Downarrow_{q * q'}^{t \cdot t'} \kappa' \vDash v}$$

$$\textbf{N-Narrow}\frac{\begin{array}{c} e \dashv \kappa \Downarrow_q^t \kappa_a \vDash v \\ e_1 \dashv \kappa_a \Downarrow_{q_1}^{t_1} \kappa_b \vDash v_1 \qquad\qquad e_2 \dashv \kappa_b \Downarrow_{q_2}^{t_2} \kappa_c \vDash v_2 \\ \mathit{sampleV}\ \kappa_c\ v_1 \Rightarrow_{q_1'}^{t_1'} \kappa_d \qquad\qquad \mathit{sampleV}\ \kappa_d\ v_2 \Rightarrow_{q_2'}^{t_2'} \kappa_e \\ \mathit{nat}_{\kappa_e}(v_1) = n_1 \qquad n_1 > 0 \qquad \mathit{nat}_{\kappa_e}(v_2) = n_2 \qquad n_2 > 0 \\ (\kappa_0, [u_1, u_2]) = \mathit{fresh}\ \kappa_e\ [\overline{T}_1, \overline{T}_2] \\ \kappa_l = \mathit{unify}\ \kappa_0\ v\ (L_{\overline{T}_1 + \overline{T}_2}\ u_1) \qquad \kappa_r = \mathit{unify}\ \kappa_0\ v\ (R_{\overline{T}_1 + \overline{T}_2}\ u_2) \\ \mathit{choose}\ n_1\ \kappa_l\ n_2\ \kappa_r \rightarrow_{q'}^{t'} i \end{array}}{e^{\overline{T}_1 + \overline{T}_2} \leftarrow (e_1^{\mathit{nat}}, e_2^{\mathit{nat}}) \dashv \kappa \Downarrow_{q * q_1 * q_2 * q_1' * q_2' * q'}^{t \cdot t_1 \cdot t_2 \cdot t_1' \cdot t_2' \cdot t'} \kappa_i \vDash v}$$

Figure 4.9: Narrowing Semantics for Non-Standard Expressions

$$\frac{SAT(\kappa_1) \quad SAT(\kappa_2)}{choose\ n\ \kappa_1\ m\ \kappa_2 \rightarrow^{[(0,2)]}_{n/(n+m)}\ l} \qquad \frac{\neg SAT(\kappa_1) \quad SAT(\kappa_2)}{choose\ n\ \kappa_1\ m\ \kappa_2 \rightarrow^{\epsilon}_{1}\ r}$$

$$\frac{SAT(\kappa_1) \quad SAT(\kappa_2)}{choose\ n\ \kappa_1\ m\ \kappa_2 \rightarrow^{[(1,2)]}_{m/(n+m)}\ r} \qquad \frac{SAT(\kappa_1) \quad \neg SAT(\kappa_2)}{choose\ n\ \kappa_1\ m\ \kappa_2 \rightarrow^{\epsilon}_{1}\ l}$$

Figure 4.10: Auxiliary Relation *choose*

$$\frac{sample\ \kappa\ u = S \quad S[m] = \kappa'}{sampleV\ \kappa\ u \Rightarrow^{[(m,|S|)]}_{1/|S|}\ \kappa'}$$

$$\frac{}{sampleV\ \kappa\ () \Rightarrow^{\epsilon}_{1}\ \kappa} \qquad \frac{sampleV\ \kappa\ v \Rightarrow^{t}_{q}\ \kappa'}{sampleV\ \kappa\ (fold_T\ v) \Rightarrow^{t}_{q}\ \kappa'}$$

$$\frac{sampleV\ \kappa\ v \Rightarrow^{t}_{q}\ \kappa'}{sampleV\ \kappa\ (L_T\ v) \Rightarrow^{t}_{q}\ \kappa'} \qquad \frac{sampleV\ \kappa\ v \Rightarrow^{t}_{q}\ \kappa'}{sampleV\ \kappa\ (R_T\ v) \Rightarrow^{t}_{q}\ \kappa'}$$

$$\frac{sampleV\ \kappa\ v_1 \Rightarrow^{t_1}_{q_1}\ \kappa_1 \quad sampleV\ \kappa_1\ v_2 \Rightarrow^{t_2}_{q_2}\ \kappa'}{sampleV\ \kappa\ (v_1, v_2) \Rightarrow^{t_1 \cdot t_2}_{q_1 * q_2}\ \kappa'}$$

Figure 4.11: Auxiliary Relation *sampleV*

when we generate the unknowns u_1 and u_2, their domains are unconstrained, but before we substitute them into an expression used as "input" to a subderivation, we unify them with the result of a narrowing derivation, which already has a finite representation in κ_a.

Rules **N-L**, **N-R**: To evaluate $L_{T_1+T_2}\ e$, we evaluate e and tag the resulting value with $L_{T_1+T_2}$, with the resulting constraint set, trace, and probability unchanged. $R_{T_1+T_2}\ e$ is handled similarly.

Rules **N-Case-L**, **N-Case-R**, **N-Case-U**: As in the pair elimination rule, we first evaluate the discriminee e to a value, which must have one of the shapes $L_T\ v_l$, $R_T\ v_r$, or $u \in \mathcal{U}$, thanks to typing. The cases for $L_T\ v_l$ (rule **N-Case-L**) and $R_T\ v_r$ (rule **N-Case-R**) are similar to **N-CasePair-P**: v_l or v_r can be directly substituted for x_l or x_r in e_l or e_r. The unknown case (**N-Case-U**) is similar to **N-CasePair-U** but a bit more complex. Once again e shares with the unknown u a type $\overline{T}_l + \overline{T}_r$ that does not contain any arrows, so we can generate fresh unknowns u_l, u_r with types $\overline{T}_l$, $\overline{T}_r$. We unify $L_{\overline{T}_l+\overline{T}_r}\ v_l$ with u to get the constraint set κ_l and $R_{\overline{T}_l+\overline{T}_r}\ v_r$ with u to get κ_r. We then use the auxiliary relation *choose* (Figure 4.10), which takes two integers n and m (here equal to 1) as well as two constraint sets (here κ_l and κ_r), to select either l or r. If exactly one of κ_l and κ_r is satisfiable, then *choose* will return the corresponding index with probability 1 and an empty trace (because no random choice were made). If both are satisfiable, then the resulting index is randomly chosen. Both outcomes are equiprobable (because of the 1 arguments to *choose*), so the probability is one half in each case. This uniform binary choice is recorded in the trace t_2 as either $(0, 2)$ or $(1, 2)$. Finally, we evaluate the expression corresponding to the chosen index, with the corresponding unknown substituted for the variable. The satisfiability checks enforce the invariant that constraint sets are satisfiable, which in turn ensures that κ_l and κ_r cannot both be unsatisfiable at the same time, since there must exist at least one valuation in κ_0 that maps u to a value (either L or R) which ensures that the corresponding unification will succeed.

Rule **N-App**: To evaluate an application $(e_0\ e_1)$, we first evaluate e_0 to $rec\ (f : T_1 \to T_2)\ x = e_2$ (since unknowns only range over arrow-free types $\overline{T}$, the result cannot be an unknown) and its argument e_1 to a value v_1. We then evaluate

the appropriately substituted body, $e_2[(rec\ (f : T_1 \to T_2)\ x = e_2)/f, v_1/x]$, and combine the various probabilities and traces appropriately.

Rule **N-After**: Rule **N-After** is similar to **N-Pair**; however, the value result of the derivation is that of the first narrowing evaluation, implementing the reverse form of sequencing described in the introduction of this section.

Rule **N-Fold**, **N-Unfold-F**, **N-Unfold-U**: Rule **N-Fold** is similar to **N-L**. **N-Unfold-F** and **N-Unfold-U** are similar to (though simpler than) **N-CasePair-P** and **N-CasePair-U**.

Rule **N-Bang**: To evaluate $!e$ we evaluate e to a value v, then use the auxiliary relation *sampleV* (Figure 4.11) to completely instantiate v, walking down the structure of v. When unknowns are encountered, *sample* is used to produce a list of constraint sets S; with probability $\frac{1}{|S|}$ (where $|S|$ is the size of the list) we can select the mth constraint set in S, for each $0 \leqslant m < |S|$.

Rule **N-Narrow** is similar to **N-Case-U**. The main difference is the "weight" arguments e_1 and e_2. These are evaluated to values v_1 and v_2, and *sampleV* is called to ensure that they are fully instantiated in all subsequent constraint sets, in particular in κ_e. The relation $nat_{\kappa_e}(v_1) = n_1$ walks down the structure of the value v_1 (like *sampleV*) and calculates the unique natural number n_1 corresponding to v_1. Specifically, when the input value is an unknown, $nat_\kappa(u) = n$ holds if $\kappa[u] = v'$ and $[\![v]\!] = n$, where the notation $[\![v]\!]$ is defined in Figure Fig. 4.6. The rest of the rule is the same as **N-Case-U**, except that the computed weights n_1 and n_2 are given as arguments to *choose* in order to shape the distribution accordingly.

Using the narrowing semantics, we can implement a more efficient method for generating valuations than the naive generate-and-test described in Section §4.2.1: instead of generating arbitrary valuations we only lazily instantiate a subset of unknowns as we encounter them. This method has the additional advantage that, if a generated valuation yields an unwanted result, the implementation can backtrack to the point of the latest choice, which can drastically improve performance [31].

Unfortunately, using the narrowing semantics in this way can lead to a lot of backtracking. To see why, consider three unknowns, u_1, u_2, and u_3, and a constraint set κ where each unknown has type `Bool` (i.e., $1 + 1$) and the domain associated with each contains both `True` and `False` (L_{1+1} () and R_{1+1} ()). Suppose we

want to generate valuations for these three unknowns such that the conjunction u_1 && u_2 && u_3 holds, where e_1 && e_2 is shorthand for *case* e_1 *of* $(L\ x \to e_2)(R\ y \to \texttt{False})$. If we attempt to evaluate the expression u_1 && u_2 && u_3 using the narrowing semantics, we first apply the **N-Case-U** rule with $e = u_1$. That means that u_1 will be unified with either L or R (applied to a fresh unknown) with equal probability, leading to a `False` result for the entire expression 50% of the time. If we choose to unify u_1 with an L, then we apply the **N-Case-U** rule again, returning either `False` or u_3 (since unknowns are values—rule **N-Base**) with equal probability. Therefore, we will have generated a desired valuation only 25% of the time; we will need to backtrack 75% of the time.

The problem here is that the narrowing semantics is agnostic to the desired result of the whole computation—we only find out at the very end that we need to backtrack. But we can do better...

4.2.4 Matching Semantics

In this section we present a *matching* semantics that takes as an additional input a *pattern* (a value not containing lambdas but possibly containing unknowns)

$$p ::= () \mid (p, p) \mid L_{\overline{T}}\ p \mid R_{\overline{T}}\ p \mid \mathit{fold}_{\overline{T}}\ p \mid u$$

and propagates this pattern backwards to guide the generation process. By allowing our semantics to look ahead in this way, we can often avoid case branches that lead to non-matching results.

The matching judgment is again a variant of big-step evaluation; it has the form

$$p \Leftarrow e \rightleftharpoons \kappa \Uparrow_q^t \kappa^?$$

where the pattern p can mention the unknowns in $U(\kappa)$ and where the metavariable $\kappa^?$ stands for an *optional* constraint set ($\varnothing$ or $\{\kappa\}$) returned by matching. Returning an option allows us to calculate the probability of backtracking by summing the q's of all failing derivations. The combined probability of failures and successes may be less than 1, because some reduction paths may diverge.

We keep the invariants from Figure 4.2.3: the input constraint set κ is well typed and so is the input expression e (with respect to an empty variable context and $U(\kappa)$); moreover κ is satisfiable, and the restriction of its denotation to the unknowns in e is finite. To these invariants we add that the input pattern p is well typed in $U(\kappa)$ and that the common type of e and p does not contain any arrows (e can still contain functions and applications internally; these are handled by calling the narrowing semantics).

The following properties are essential to maintaining these invariants. Whenever the output option has the form $\{\kappa'\}$, then κ' is satisfiable. This is easily ensured by checking the satisfiability of candidate constraint sets and outputting $\varnothing$ if they are not satisfiable. Moreover, when the output has the form $\{\kappa'\}$, then all the unknowns of p have finite denotations in κ' (despite them not necessarily having finite denotations in the input constraint set κ).

The evaluation relation appears in Figures 4.12 (standard constructs) and 4.15 (novel Luck constructs). Additional rules concerning failure propagating cases appear in Figure 4.14, while match rules that deal with discriminees containing arrow types appear in Figure 4.13. Most of them are largely similar to the narrowing rules, only introducing unifications with target patterns in key places. Several of them rely on the narrowing semantics defined previously.

Rule **M-Base**: To generate valuations for a unit value or an unknown, we unify v and the target pattern p under the input constraint set κ. Unlike **N-Base**, there is no case for functions, since the expression being evaluated must have a non-function type.

Rules **M-Pair**, **M-Pair-Fail**: To evaluate (e_1, e_2), where e_1 and e_2 have types $\overline{T}_1$ and $\overline{T}_2$, we first generate fresh unknowns u_1 and u_2 with these types. We unify the pair (u_1, u_2) with the target pattern p, obtaining a new constraint set κ'. We then proceed as in **N-Pair**, evaluating e_1 against pattern u_1 and e_2 against u_2, threading constraint sets and accumulating traces and probabilities. **M-Pair** handles the case where the evaluation of e_1 succeeds, yielding a constraint set $\{\kappa_1\}$, while **M-Pair-Fail** handles failure: if evaluating e_1 yields $\varnothing$, the whole computation immediately yields $\varnothing$ as well; e_2 is not evaluated, and the final trace and probability are t_1 and q_1.

$$\textbf{M-Base}\ \frac{v = () \ \vee \ v \in \mathcal{U} \qquad \kappa' = \mathit{unify}\ \kappa\ v\ p}{p \Leftarrow v \dashv \kappa \Uparrow_1^{\epsilon}\ \mathit{if}\ \mathit{SAT}(\kappa')\ \mathit{then}\ \{\kappa'\}\ \mathit{else}\ \varnothing}$$

$$\textbf{M-Pair}\ \frac{\begin{array}{c}(\kappa', [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2] \\ \kappa_0 = \mathit{unify}\ \kappa'\ (u_1, u_2)\ p \\ u_1 \Leftarrow e_1 \dashv \kappa_0 \Uparrow_{q_1}^{t_1} \{\kappa_1\} \qquad u_2 \Leftarrow e_2 \dashv \kappa_1 \Uparrow_{q_2}^{t_2} \kappa_2^?\end{array}}{p \Leftarrow (e_1^{\overline{T}_1}, e_2^{\overline{T}_2}) \dashv \kappa \Uparrow_{q_1 * q_2}^{t_1 \cdot t_2} \kappa_2^?}$$

$$\textbf{M-CasePair}\ \frac{\begin{array}{c}(\kappa_a, [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2] \\ (u_1, u_2) \Leftarrow e \dashv \kappa_a \Uparrow_{q_1}^{t_1} \{\kappa_b\} \\ p \Leftarrow e'[u_1/x, u_2/y] \dashv \kappa_b \Uparrow_{q_2}^{t_2} \kappa^?\end{array}}{p \Leftarrow \mathit{case}\ e^{\overline{T}_1 \times \overline{T}_2}\ \mathit{of}\ (x, y) \to e' \dashv \kappa \Uparrow_{q_1 * q_2}^{t_1 \cdot t_2} \kappa^?}$$

$$\textbf{M-L-Sat}\ \frac{\begin{array}{c}(\kappa_1, u) = \mathit{fresh}\ \kappa\ \overline{T}_1 \\ \kappa_2 = \mathit{unify}\ \kappa_1\ (L_{\overline{T}_1 + \overline{T}_2}\ u)\ p \\ \mathit{SAT}(\kappa_2) \qquad u \Leftarrow e \dashv \kappa_2 \Uparrow_q^t \kappa^?\end{array}}{p \Leftarrow L_{\overline{T}_1 + \overline{T}_2}\ e \dashv \kappa \Uparrow_q^t \kappa^?}$$

$$\textbf{M-R-Sat}\ \frac{\begin{array}{c}(\kappa_1, u) = \mathit{fresh}\ \kappa\ \overline{T}_2 \\ \kappa_2 = \mathit{unify}\ \kappa_1\ (R_{\overline{T}_1 + \overline{T}_2}\ u)\ p \\ \mathit{SAT}(\kappa_2) \qquad u \Leftarrow e \dashv \kappa_2 \Uparrow_q^t \kappa^?\end{array}}{p \Leftarrow R_{\overline{T}_1 + \overline{T}_2}\ e \dashv \kappa \Uparrow_q^t \kappa^?}$$

$$\textbf{M-App}\ \frac{\begin{array}{c}e_0 \dashv \kappa \Downarrow_{q_0}^{t_0} \kappa_0 \vDash (\mathit{rec}\ f\ x = e_2) \\ e_1 \dashv \kappa_0 \Downarrow_{q_1}^{t_1} \kappa' \vDash v_1 \\ p \Leftarrow e_2[(\mathit{rec}\ f\ x = e_2)/f, v_1/x] \dashv \kappa' \Uparrow_{q_2}^{t_2} \kappa^?\end{array}}{p \Leftarrow (e_0\ e_1) \dashv \kappa \Uparrow_{q_0 * q_1 * q_2}^{t_0 \cdot t_1 \cdot t_2} \kappa^?}$$

$$\textbf{M-Fold}\ \frac{\begin{array}{c}(\kappa_1, u) = \mathit{fresh}\ \kappa\ \overline{T}[\mu X.\ \overline{T}/X] \\ \kappa_2 = \mathit{unify}\ \kappa_1\ (\mathit{fold}_{\mu X.\ \overline{T}}\ u)\ p \\ u \Leftarrow e \dashv \kappa_2 \Uparrow_q^t \kappa^?\end{array}}{p \Leftarrow \mathit{fold}_{\mu X.\ \overline{T}}\ e \dashv \kappa \Uparrow_q^t \kappa^?} \qquad \textbf{M-Unfold}\ \frac{(\mathit{fold}_{\mu X.\ \overline{T}}\ p) \Leftarrow e \dashv \kappa \Uparrow_q^t \kappa^?}{p \Leftarrow \mathit{unfold}_{\mu X.\ \overline{T}}\ e \dashv \kappa \Uparrow_q^t \kappa^?}$$

Figure 4.12: Matching Semantics of Standard Core Luck Constructs

$$\textbf{M-CasePair-Fun}\ \frac{\begin{array}{c} T_1 \notin \overline{T} \vee T_2 \notin \overline{T} \\ e \rightleftharpoons \kappa \Downarrow_{q_1}^{t_1} \kappa_1 \vDash (v_1, v_2) \\ p \Leftarrow e'[v_1/x, v_2/y] \rightleftharpoons \kappa_1 \Uparrow_{q_2}^{t_2} \kappa_2^? \end{array}}{p \Leftarrow \mathit{case}\ e^{T_1 \times T_2}\ \mathit{of}\ (x, y) \to e' \rightleftharpoons \kappa \Uparrow_{q_1 * q_2}^{t_1 \cdot t_2} \kappa_2^?}$$

$$\textbf{M-Case-L-Fun}\ \frac{\begin{array}{c} T_1 \notin \overline{T} \vee T_2 \notin \overline{T} \\ e \rightleftharpoons \kappa \Downarrow_{q_1}^{t_1} \kappa_1 \vDash L_{T_1+T_2}\ v_1 \\ p \Leftarrow e_1[v_1/x_l] \rightleftharpoons \kappa_1 \Uparrow_{q_1'}^{t_1'} \kappa^? \end{array}}{p \Leftarrow \mathit{case}\ e^{T_1+T_2}\ \mathit{of}\ (L\ x_l \to e_1)(R\ x_r \to e_2) \rightleftharpoons \kappa \Uparrow_{q_1 * q_1'}^{t_1 \cdot t_1'} \kappa^?}$$

$$\textbf{M-Case-R-Fun}\ \frac{\begin{array}{c} T_1 \notin \overline{T} \vee T_2 \notin \overline{T} \\ e \rightleftharpoons \kappa \Downarrow_{q_1}^{t_1} \kappa_1 \vDash R_{T_1+T_2}\ v_2 \\ p \Leftarrow e_2[v_2/x_r] \rightleftharpoons \kappa_1 \Uparrow_{q_1'}^{t_1'} \kappa^? \end{array}}{p \Leftarrow \mathit{case}\ e^{T_1+T_2}\ \mathit{of}\ (L\ x_l \to e_1)(R\ x_r \to e_2) \rightleftharpoons \kappa \Uparrow_{q_1 * q_1'}^{t_1 \cdot t_1'} \kappa^?}$$

Figure 4.13: Matching Semantics for Function Cases

$$\textbf{M-Pair-Fail}\ \frac{\begin{array}{c}(\kappa', [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2] \\ \kappa_0 = \mathit{unify}\ \kappa'\ (u_1, u_2)\ p \\ u_1 \Leftarrow e_1 \dashv \kappa_0 \Uparrow_{q_1}^{t_1} \varnothing\end{array}}{p \Leftarrow (e_1^{\overline{T}_1}, e_2^{\overline{T}_2}) \dashv \kappa \Uparrow_{q_1}^{t_1} \varnothing}$$

$$\textbf{M-CasePair-Fail}\ \frac{\begin{array}{c}([u_1, u_2], \kappa_0) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2] \\ (u_1, u_2) \Leftarrow e \dashv \kappa_0 \Uparrow_{q_1}^{t_1} \varnothing\end{array}}{p \Leftarrow \mathit{case}\ e^{\overline{T}_1 \times \overline{T}_2}\ \mathit{of}\ (x, y) \rightarrow e' \dashv \kappa \Uparrow_{q_1}^{t_1} \varnothing}$$

$$\textbf{M-After-Fail}\ \frac{p \Leftarrow e_1 \dashv \kappa \Uparrow_{q_1}^{t_1} \varnothing}{p \Leftarrow e_1\ ;\ e_2 \dashv \kappa \Uparrow_{q_1}^{t_1} \varnothing}$$

$$\textbf{M-L-UnSat}\ \frac{\begin{array}{c}(\kappa_1, u) = \mathit{fresh}\ \kappa\ \overline{T}_1 \\ \kappa_2 = \mathit{unify}\ \kappa_1\ (L_{\overline{T}_1 + \overline{T}_2}\ u)\ p \\ \neg SAT(\kappa_2)\end{array}}{p \Leftarrow L_{\overline{T}_1 + \overline{T}_2}\ e \dashv \kappa \Uparrow_1^{\epsilon} \varnothing}$$

$$\textbf{M-R-UnSat}\ \frac{\begin{array}{c}(\kappa_1, u) = \mathit{fresh}\ \kappa\ \overline{T}_2 \\ \kappa_2 = \mathit{unify}\ \kappa_1\ (R_{\overline{T}_1 + \overline{T}_2}\ u)\ p \\ \neg SAT(\kappa_2)\end{array}}{p \Leftarrow R_{\overline{T}_1 + \overline{T}_2}\ e \dashv \kappa \Uparrow_1^{\epsilon} \varnothing}$$

Figure 4.14: Failure Propagation for Matching Semantics

$$\textbf{M-After}\ \frac{p \Leftarrow e_1 \rightleftharpoons \kappa \Uparrow_{q_1}^{t_1} \{\kappa_1\} \qquad e_2 \rightleftharpoons \kappa_1 \Downarrow_{q_2}^{t_2} \kappa_2 \vDash v}{p \Leftarrow e_1 \,;\, e_2 \rightleftharpoons \kappa \Uparrow_{q_1 * q_2}^{t_1 \cdot t_2} \{\kappa_2\}}$$

$$\textbf{M-Bang}\ \frac{\begin{array}{c} p \Leftarrow e \rightleftharpoons \kappa \Uparrow_{q_1}^{t_1} \{\kappa_1\} \\ sampleV\ \kappa_1\ p \Rightarrow_{q_2}^{t_2} \kappa' \end{array}}{p \Leftarrow !e \rightleftharpoons \kappa \Uparrow_{q_1 * q_2}^{t_1 \cdot t_2} \{\kappa'\}}$$

$$\textbf{M-Bang-Fail}\ \frac{p \Leftarrow e \rightleftharpoons \kappa \Uparrow_{q_1}^{t_1} \varnothing}{p \Leftarrow !e \rightleftharpoons \kappa \Uparrow_{q_1}^{t_1} \varnothing}$$

$$\textbf{M-Narrow}\ \frac{\begin{array}{c} p \Leftarrow e \rightleftharpoons \kappa \Uparrow_{q}^{t} \{\kappa_a\} \\ e_1 \rightleftharpoons \kappa_a \Downarrow_{q_1}^{t_1} \kappa_b \vDash v_1 \qquad e_2 \rightleftharpoons \kappa_b \Downarrow_{q_2}^{t_2} \kappa_c \vDash v_2 \\ sampleV\ \kappa_c\ v_1 \Rightarrow_{q_1'}^{t_1'} \kappa_d \qquad sampleV\ \kappa_d\ v_2 \Rightarrow_{q_2'}^{t_2'} \kappa_e \\ nat_{\kappa_e}(v_1) = n_1 \quad n_1 > 0 \quad nat_{\kappa_e}(v_2) = n_2 \quad n_2 > 0 \\ (\kappa_0, [u_1, u_2]) = fresh\ \kappa_e\ [\overline{T}_1, \overline{T}_2] \\ \kappa_l = unify\ \kappa_0\ p\ (L_{\overline{T}_1 + \overline{T}_2}\ u_1) \\ \kappa_r = unify\ \kappa_0\ p\ (R_{\overline{T}_1 + \overline{T}_2}\ u_2) \\ choose\ n_1\ \kappa_l\ n_2\ \kappa_r \rightarrow_{q'}^{t'} i \end{array}}{p \Leftarrow e^{\overline{T}_1 + \overline{T}_2} \leftarrow (e_1^{nat}, e_2^{nat}) \rightleftharpoons \kappa \Uparrow_{q * q_1 * q_2 * q_1' * q_2' * q'}^{t \cdot t_1 \cdot t_2 \cdot t_1' \cdot t_2' \cdot t'} \{\kappa_i\}}$$

$$\textbf{M-Narrow-Fail}\ \frac{p \Leftarrow e \rightleftharpoons \kappa \Uparrow_{q}^{t} \varnothing}{p \Leftarrow e^{\overline{T}_1 + \overline{T}_2} \leftarrow (e_1^{nat}, e_2^{nat}) \rightleftharpoons \kappa \Uparrow_{q}^{t} \varnothing}$$

Figure 4.15: Matching Semantics of Nonstandard Core Luck Constructs

Rules **M-CasePair**, **M-CasePair-Fail**, **M-CasePair-Fun**: If the type of the discriminee e contains function types (**M-CasePair-Fun**), we narrow e to a pair and substitute its components as in **N-CasePair-P**, but then we evaluate the resulting expression against the original target pattern p. Otherwise e has a type of form $\overline{T}_1 \times \overline{T}_2$ and we proceed as in **N-CasePair-U** with a few differences. The unknowns u_1 and u_2 are introduced before the evaluation of e to provide a target pattern (u_1, u_2). If the evaluation succeeds in yielding $\{\kappa_b\}$ (**M-CasePair**) we proceed to substitute u_1 and u_2 (that now have a finite domain as all pattern unknowns at the resulting constraint sets). If instead evaluation of e yields $\varnothing$ (**M-CasePair-Fail**), the whole computation returns $\varnothing$ immediately.

Rules **M-L-Sat**, **M-R-Sat**, **M-L-UnSat**, **M-R-UnSat**: To evaluate $L_{\overline{T}_1+\overline{T}_2}\ e$, we generate an unknown u of type $\overline{T}_1$ and unify $L_{\overline{T}_1+\overline{T}_2}\ u$ with the target pattern p. If the constraint set obtained is satisfiable (**M-L-Sat**), we simply evaluate e against the pattern u. Otherwise (**M-L-UnSat**) we immediately return $\varnothing$. The same goes for R.

Rules **M-App**, **M-After**: To evaluate an application $e_0\ e_1$, we use the narrowing semantics to reduce e_0 to $rec\ f\ x = e_2$ and e_1 to a value v_1, then evaluate $e_2[(rec\ f\ x = e_2)/f, v_2/x]$ against the original target pattern p in the matching semantics. In this rule we cannot use a pattern during the evaluation of e_1: we do not have any candidates! This is the main reason for introducing the sequencing operator as a primitive $e_1\ ;\ e_2$ instead of encoding it using lambda abstractions. In **M-After**, we evaluate e_1 against he target pattern p and then evaluate e_2 using narrowing, just for its side effects. If we used lambdas to encode sequencing, e_1 would be narrowed instead, which is not what we want.

Rules **M-Fold**, **M-Unfold**: **M-Fold** is similar to **M-Pair**, only simpler. To evaluate $unfold_{\mu X.\ \overline{T}}\ e$ with pattern p, **M-Unfold** simply evaluates e with the pattern $fold_{\mu X.\ \overline{T}}\ p$.

Rules **M-Bang**, **M-Bang-Fail**: This rule is very similar to **N-Bang**. We first evaluate e against pattern p. If that succeeds we proceed to use the same auxiliary relation $sampleV$ as in **N-Bang** (defined in Figure 4.11). Otherwise, the whole computation returns $\varnothing$.

Rules **M-Narrow**, **M-Narrow-Fail**: Like in **M-Bang**, we propagate the pat-

tern p and evaluate e against it. After checking that the resulting constraint set option is not $\varnothing$, we proceed exactly as in **N-Narrow**.

Rules **M-Case-L-Fun**, **M-Case-R-Fun**: If the type of the discriminee e contains function types (meaning it cannot be written as $\overline{T}_1 + \overline{T}_2$), we proceed as in **N-Case-L** and **N-Case-R**, except in the final evaluation we match the expression against p.

The interesting rules are the ones for *case* when the type of the scrutinee does not contain functions. For these rules, we can actually use the patterns to guide the generation that occurs during the evaluation of the scrutinee as well. We use them to model the constraint solving behavior: instead of choosing which branch to follow with some probability (50% in **N-Case-U**), we evaluate both branches, just like a constraint solver would exhaustively search the entire domain.

Before looking at the rules in detail, we need to extend the constraint set interface with two new functions:

$$\begin{array}{lll} \mathit{rename} & :: & \mathcal{U}^* \to \mathcal{C} \to \mathcal{C} \\ \mathit{union} & :: & \mathcal{C} \to \mathcal{C} \to \mathcal{C} \end{array}$$

The *rename* operation freshens a constraint set by replacing all the unknowns in a given sequence with freshly generated ones (of the same type). The *union* of two constraint sets intuitively denotes the union of their corresponding denotations.

The four *case* rules with function-free types appear in Figure 4.16. We independently evaluate e against both an L pattern and an R pattern. If both of them yield failure, then the whole evaluation yields failure (**M-Case-4**). If exactly one succeeds, we evaluate just the corresponding branch (**M-Case-2** or **M-Case-3**). If both succeed (**M-Case-1**), we evaluate both branch bodies and combine the results with *union*. We use *rename* to avoid conflicts, since we may generate the same fresh unknowns while independently computing $\kappa_a^?$ and $\kappa_b^?$.

If desired, the user can ensure that only one branch will be executed by using an instantiation expression before the *case* is reached. Since e will then begin with a concrete constructor, only one of the evaluations of e against the patterns L and R will succeed, and only the corresponding branch will be executed.

The **M-Case-1** rule is the second place where the need for finiteness of the restriction of κ to the input expression e arises. In order for the semantics to

$$\frac{\begin{array}{c}(\kappa_0, [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2] \\ (L_{\overline{T}_1+\overline{T}_2}\ u_1) \Leftarrow e \dashv \kappa_0 \Uparrow_{q_1}^{t_1} \{\kappa_1\} \\ (R_{\overline{T}_1+\overline{T}_2}\ u_2) \Leftarrow e \dashv \kappa_0 \Uparrow_{q_2}^{t_2} \{\kappa_2\} \\ p \Leftarrow e_1[u_1/x_l] \dashv \kappa_1 \Uparrow_{q_1'}^{t_1'} \kappa_a^? \qquad p \Leftarrow e_2[u_2/y_r] \dashv \kappa_2 \Uparrow_{q_2'}^{t_2'} \kappa_b^? \\ \kappa^? = \mathit{combine}\ \kappa_0\ \kappa_a^?\ \kappa_b^? \end{array}}{p \Leftarrow \mathit{case}\ e^{\overline{T}_1+\overline{T}_2}\ \mathit{of}\ (L\ x_l \to e_1)(R\ y_r \to e_2) \dashv \kappa \Uparrow_{q_1*q_2*q_1'*q_2'}^{t_1 \cdot t_2 \cdot t_1' \cdot t_2'} \kappa^?}\ \textbf{M-Case-1}$$

where $\mathit{combine}\ \kappa\ \varnothing\ \varnothing = \varnothing$
$\mathit{combine}\ \kappa\ \{\kappa_1\}\ \varnothing = \{\kappa_1\}$
$\mathit{combine}\ \kappa\ \varnothing\ \{\kappa_2\} = \{\kappa_2\}$
$\mathit{combine}\ \kappa\ \{\kappa_1\}\ \{\kappa_2\} =$
$\{\mathit{union}\ \kappa_1\ (\mathit{rename}\ (U(\kappa_1)\text{-}U(\kappa))\ \kappa_2)\}$

$$\frac{\begin{array}{c}(\kappa_0, [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2] \\ (L_{\overline{T}_1+\overline{T}_2}\ u_1) \Leftarrow e \dashv \kappa_0 \Uparrow_{q_1}^{t_1} \varnothing \\ (R_{\overline{T}_1+\overline{T}_2}\ u_2) \Leftarrow e \dashv \kappa_0 \Uparrow_{q_2}^{t_2} \{\kappa_2\} \\ p \Leftarrow e_2[u_2/y] \dashv \kappa_2 \Uparrow_{q_2'}^{t_2'} \kappa_b^? \end{array}}{p \Leftarrow \mathit{case}\ e^{\overline{T}_1+\overline{T}_2}\ \mathit{of}\ (L\ x \to e_1)(R\ y \to e_2) \dashv \kappa \Uparrow_{q_1*q_2*q_2'}^{t_1 \cdot t_2 \cdot t_2'} \kappa_b^?}\ \textbf{M-Case-2}$$

$$\frac{\begin{array}{c}(\kappa_0, [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2] \\ (L_{\overline{T}_1+\overline{T}_2}\ u_1) \Leftarrow e \dashv \kappa_0 \Uparrow_{q_1}^{t_1} \{\kappa_1\} \\ (R_{\overline{T}_1+\overline{T}_2}\ u_2) \Leftarrow e \dashv \kappa_0 \Uparrow_{q_2}^{t_2} \varnothing \\ p \Leftarrow e_1[u_1/x] \dashv \kappa_1 \Uparrow_{q_1'}^{t_1'} \kappa_a^? \end{array}}{p \Leftarrow \mathit{case}\ e^{\overline{T}_1+\overline{T}_2}\ \mathit{of}\ (L\ x \to e_1)(R\ y \to e_2) \dashv \kappa \Uparrow_{q_1*q_2*q_1'}^{t_1 \cdot t_2 \cdot t_1'} \kappa_a^?}\ \textbf{M-Case-3}$$

$$\frac{\begin{array}{c}(\kappa_0, [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2] \\ (L_{\overline{T}_1+\overline{T}_2}\ u_1) \Leftarrow e \dashv \kappa_0 \Uparrow_{q_1}^{t_1} \varnothing \\ (R_{\overline{T}_1+\overline{T}_2}\ u_2) \Leftarrow e \dashv \kappa_0 \Uparrow_{q_2}^{t_2} \varnothing \end{array}}{p \Leftarrow \mathit{case}\ e^{\overline{T}_1+\overline{T}_2}\ \mathit{of}\ (L\ x \to e_1)(R\ y \to e_2) \dashv \kappa \Uparrow_{q_1*q_2}^{t_1 \cdot t_2} \varnothing}\ \textbf{M-Case-4}$$

Figure 4.16: Matching Semantics for Constraint-Solving *case*

terminate in the presence of (terminating) recursive calls, it is necessary that the domain be finite. To see this, consider a simple recursive predicate that holds for every number:

$$\begin{array}{l} \mathit{rec}\ (f : \mathit{nat} \to \mathit{bool})\ u = \\ \quad \mathit{case}\ \mathit{unfold}_{\mathit{nat}}\ u\ \mathit{of}\ (L\ x \to \texttt{True})(R\ y \to (f\ y)) \end{array}$$

Even though f terminates in the predicate semantics for every input u, if we allow a constraint set to map u to the infinite domain of all natural numbers, the matching semantics will not terminate. While this finiteness restriction feels a bit unnatural, we have not found it to be a problem in practice—see Section 4.4.

4.2.5 Example

To show how all this works, let's trace the main steps of the matching derivations of two given expressions against the pattern `True` in a given constraint set. We will also extract probability distributions about optional constraint sets from these derivations.

We are going to evaluate $A := (0 < u \mathbin{\&\&} u < 4)\ ;\ !u$ and $B := (0 < u\ ;\ !u) \mathbin{\&\&} u < 4$ against the pattern `True` in a constraint set κ, in which u is independent from other unknowns and its possible values are $0, ..., 9$. Similar expressions were introduced as examples in Section 4.1; the results we obtain here confirm the intuitive explanation given there.

Recall that the conjunction expression $e_1 \mathbin{\&\&} e_2$ is just syntactic sugar for $\mathit{case}\ e_1\ \mathit{of}\ (L\ a \to e_2)(R\ b \to \texttt{False})$, and that we are using a standard Peano encoding of naturals: $\mathit{nat} = \mu X.\ 1 + X$. We elide folds for brevity. The inequality $a < b$ can be encoded as $\mathit{lt}\ a\ b$, where:

$$\begin{array}{l} \mathit{lt} = \mathit{rec}\ (f : \mathit{nat} \to \mathit{nat} \to \mathit{bool})\ x = \mathit{rec}\ (g : \mathit{nat} \to \mathit{bool})\ y = \\ \quad \mathit{case}\ y\ \mathit{of}\quad (L\ \_ \to \texttt{False}) \\ \qquad\qquad\quad (R\ y_R \to \mathit{case}\ x\ \mathit{of}\quad (L\ \_ \to \texttt{True}) \\ \qquad\qquad\qquad\qquad\qquad\quad (R\ x_R \to f\ x_R\ y_R)) \end{array}$$

Many rules introduce fresh unknowns, many of which are irrelevant: they might be directly equivalent to some other unknown, or there might not exist any reference to them. We abusively use the same variable for two constraint sets which differ only in the addition of a few irrelevant variables to one of them.

Evaluation of A We first derive $\texttt{True} \Leftarrow (0 < u) \rightleftharpoons \kappa \Uparrow_1^\epsilon \{\kappa_0\}$. Since in the desugaring of $0 < u$ as an application lt is already in *rec* form and both 0 and u are values, the constraint set after the narrowing calls of **M-App** will stay unchanged. We then evaluate *case* u *of* $(L \_ \to \texttt{False})(R\ y_R \to ...)$. Since the domain of u contains both zero and non-zero elements, unifying u with $L_{1+nat}\ u_1$ and $R_{1+nat}\ u_2$ (**M-Base**) will produce some non-empty constraint sets. Therefore, rule **M-Case-1** applies. Since the body of the left hand side of the match is `False`, the result of the left derivation in **M-Case-1** is $\varnothing$ and in the resulting constraint set κ_0 the domain of u is $\{1, ..., 9\}$.

Next, we turn to $\texttt{True} \Leftarrow (0 < u \mathbin{\&\&} u < 4) \rightleftharpoons \kappa \Uparrow_1^\epsilon \{\kappa_1\}$, where, by a similar argument following the recursion, the domain of u in κ_1 is $\{1, 2, 3\}$. There are 3 possible narrowing-semantics derivations for $!u$: (1) $!u \rightleftharpoons \kappa_1 \Downarrow_{1/3}^{[(0,3)]} \kappa_1^A \vDash u$, (2) $!u \rightleftharpoons \kappa_1 \Downarrow_{1/3}^{[(1,3)]} \kappa_2^A \vDash u$, and (3) $!u \rightleftharpoons \kappa_1 \Downarrow_{1/3}^{[(2,3)]} \kappa_3^A \vDash u$, where the domain of u in κ_i^A is $\{i\}$. (We have switched to narrowing-semantics judgments because of the rule **M-After**.) Therefore all the possible derivations for $A = (0 < u \mathbin{\&\&} u < 4)\ ;\ !u$ matching `True` in κ are:

$$\texttt{True} \Leftarrow A \rightleftharpoons \kappa \Uparrow_{1/3}^{[(i-1,3)]} \{\kappa_i^A\} \qquad \text{for } i \in \{1, 2, 3\}$$

From the set of possible derivations, we can extract a probability distribution: for each resulting optional constraint set, we sum the probabilities of each of the traces that lead to this result. Thus the probability distribution associated with $\texttt{True} \Leftarrow A \rightleftharpoons \kappa$ is

$$[\{\kappa_1^A\} \mapsto \frac{1}{3}; \quad \{\kappa_2^A\} \mapsto \frac{1}{3}; \quad \{\kappa_3^A\} \mapsto \frac{1}{3}].$$

Evaluation of B The evaluation of $0 < u$ is the same as before, after which we narrow $!u$ directly in κ_0 and there are 9 possibilities: $!u \rightleftharpoons \kappa_0 \Downarrow_{1/9}^{[(i-1,9)]} \kappa_i^B \vDash u$

for each $i \in \{1, ..., 9\}$, where the domain of u in κ_i^B is $\{i\}$. Then we evaluate $\texttt{True} \Leftarrow u < 4 \dashv \kappa_i^B$: if i is 1, 2 or 3 this yields $\{\kappa_i^B\}$; if $i > 3$ this yields a failure $\varnothing$. Therefore the possible derivations for $B = (0 < u \,;\, !u)$ && $u < 4$ are:

$$\texttt{True} \Leftarrow B \dashv \kappa \Uparrow_{1/9}^{[(i-1,9)]} \{\kappa_i^B\} \qquad \text{for } i \in \{1, 2, 3\}$$
$$\texttt{True} \Leftarrow B \dashv \kappa \Uparrow_{1/9}^{[(i-1,9)]} \varnothing \qquad \text{for } i \in \{4, ..., 9\}$$

We can again compute the corresponding probability distribution:

$$[\{\kappa_1^B\} \mapsto \frac{1}{9}; \quad \{\kappa_2^B\} \mapsto \frac{1}{9}; \quad \{\kappa_3^B\} \mapsto \frac{1}{9}; \quad \varnothing \mapsto \frac{2}{3}]$$

Note that if we were just recording the probability of an execution and not its trace, we would not know that there are six distinct executions leading to $\varnothing$ with probability $\frac{1}{9}$, so we would not be able to compute its total probability.

The probability associated with $\varnothing$ (0 for A, 2/3 for B) is the probability of backtracking. As stressed in Section 4.1, A is much better than B in terms of backtracking—i.e., it is more efficient in this case to instantiate u only after all the constraints on its domain have been recorded. For a more formal treatment of backtracking strategies in Luck using Markov Chains, see [46].

4.3 Metatheory

We close our discussion of Core Luck by stating and proving some key properties. Intuitively, we show that, when we evaluate an expression e against a pattern p in the presence of a constraint set κ, we can only remove valuations from the denotation of κ (*decreasingness*), any derivation in the generator semantics corresponds to an execution in the predicate semantics (*soundness*), and every valuation that matches p will be found in the denotation of the resulting constraint set of some derivation (*completeness*).

Since we have two flavors of generator semantics, narrowing and matching, we also present these properties in two steps. First, we present the properties for the narrowing semantics; their proofs have been verified using Coq. Then we present

the properties for the matching semantics; for these, we have only paper proofs, but these proofs are quite similar to the narrowing ones (the only real difference is the case rule). Before that, however, we need to present the formal specification of the various constraint set operations.

4.3.1 Constraint Set Specification

We introduce one extra abstraction, the *domain* of a constraint set κ, written $dom(\kappa)$. This domain corresponds to the unknowns in a constraint set that actually have bindings in $[\![\kappa]\!]$. For example, when we generate a fresh unknown u from κ, u does not appear in the domain of κ; it only appears in the denotation after we use it in a unification. The domain of κ is a subset of the set of keys of $U(\kappa)$.

When we write that for a valuation and constraint set $\sigma \in [\![\kappa]\!]$, it also implies that the unknowns that have bindings in σ are exactly the unknowns that have bindings in $[\![\kappa]\!]$, i.e., in $dom(\kappa)$. We use the overloaded notation $\sigma|_x$ to denote the restriction of σ to x, where x is either a set of unknowns or another valuation (where σ is restricted to the domain of x).

The following straightforward lemma relates the two restrictions: [2] if we restrict a valuation σ' to the domain of a constraint set κ, the resulting valuation is equivalent to restricting σ' to any valuation $\sigma \in [\![\kappa]\!]$.

Lemma 4.3.1.1. $\sigma \in \kappa \;\Rightarrow\; \sigma'|_{dom(\kappa)} \equiv \sigma'|_{\sigma}$

Ordering We introduce an ordering on constraint sets: two constraints sets are *ordered* ($\kappa_1 \leqslant \kappa_2$) if $dom(\kappa_2) \subseteq dom(\kappa_1)$ and for all valuations $\sigma \in [\![\kappa_1]\!]$, $\sigma|_{dom(\kappa_2)} \in [\![\kappa_2]\!]$. Right away we can prove that $\leqslant$ is reflexive and transitive, using Lemma 4.3.1.1 and basic set properties.

[2] All the definitions in this section are implicitly universally quantified over the free variables appearing the formulas.

Specification of fresh

$$(\kappa', u) = \mathit{fresh}\ \kappa\ T \Rightarrow \begin{cases} u \notin U(\kappa) \\ U(\kappa') = U(\kappa) \oplus (u \mapsto T) \\ [\![\kappa']\!] = [\![\kappa]\!] \end{cases}$$

Intuitively, when we generate a fresh unknown u of type T from κ, u is really fresh for κ, meaning $U(\kappa)$ does not have a type binding for it. The resulting constraint set κ' has an extended unknown typing map, where u maps to T and its denotation remains unchanged. That means that $\mathit{dom}(\kappa') = \mathit{dom}(\kappa)$.

Based on this specification, we can easily prove that κ' is smaller than κ, the generated unknowns are not contained in any valuation in $[\![\kappa]\!]$ and that κ' is well typed.

Lemma 4.3.1.2 (fresh_ordered)**.**

$$(\kappa', u) = \mathit{fresh}\ \kappa\ T \ \Rightarrow\ \kappa' \leqslant \kappa$$

Lemma 4.3.1.3 (fresh_for_valuation)**.**

$$(\kappa', u) = \mathit{fresh}\ \kappa\ T \ \Rightarrow\ \forall \sigma.\ \sigma \in [\![\kappa]\!] \ \Rightarrow\ u \notin \sigma$$

Lemma 4.3.1.4 (fresh_types)**.**

$$(\kappa', u) = \mathit{fresh}\ \kappa\ T \ \Rightarrow\ (\vdash \kappa \ \Rightarrow\ \vdash \kappa')$$

Specification of sample

$$\kappa' \in \mathit{sample}\ \kappa\ u \Rightarrow \begin{cases} U(\kappa') = U(\kappa) \\ SAT(\kappa') \\ \exists v.\ [\![\kappa']\!] = \{\, \sigma \mid \sigma \in [\![\kappa]\!],\ \sigma(u) = v \,\} \end{cases}$$

When we sample u in a constraint set κ and obtain a list, for every member constraint set κ', the typing map of κ remains unchanged and all of the valuations that remain in the denotation of κ' are the ones that mapped to some specific

value v in κ. Clearly, the domain of κ remains unchanged. We also require a completeness property from *sample*, namely that if we have a valuation $\sigma \in [\![\kappa]\!]$ where $\sigma(u) = v$ for some u, v, then σ is in some member κ' of the result:

$$\left.\begin{array}{l}\sigma(u) = v \\ \sigma \in [\![\kappa]\!]\end{array}\right\} \Rightarrow \exists \kappa'. \left\{\begin{array}{l}\sigma \in [\![\kappa']\!] \\ \kappa' \in \mathit{sample}\ \kappa\ u\end{array}\right.$$

We can prove similar lemmas as in *fresh*: ordering and preservation. In addition, we can show that if some unknown is a singleton in κ, it remains a singleton in κ'. This is necessary for the proof of narrowing expressions.

Lemma 4.3.1.5 (sample_ordered)**.**

$$\kappa' \in \mathit{sample}\ \kappa\ u\ \Rightarrow\ \kappa' \leqslant \kappa$$

Lemma 4.3.1.6 (sample_types)**.**

$$\kappa' \in \mathit{sample}\ \kappa\ u\ \Rightarrow\ (\vdash \kappa\ \Rightarrow\ \vdash \kappa')$$

Lemma 4.3.1.7 (sample_preserves_singleton)**.**

$$\kappa[u'] \neq \varnothing\ \wedge\ \kappa' \in \mathit{sample}\ \kappa\ u\ \Rightarrow\ \kappa'[u'] = \kappa[u']$$

Finally, we can lift all of these properties to $\mathit{sample}V$ by simple induction, using this spec to discharge the base case.

Specification of unify

$$U(\mathit{unify}\ \kappa\ v_1\ v_2) = U(\kappa)$$
$$[\![\mathit{unify}\ \kappa\ v_1\ v_2]\!] = \{\, \sigma \in [\![\kappa]\!] \mid \sigma(v_1) = \sigma(v_2) \,\}$$

When we unify in a constraint set κ two (well-typed for κ) values v_1 and v_2, the typing map remains unchanged while the denotation of the result contains just the valuations from κ that when substituted into v_1 and v_2 make them equal. The

domain of κ' is the union of the domain of κ and the unknowns in v_1, v_2.

Once again, we can prove ordering and typing lemmas.

Lemma 4.3.1.8 (unify_ordered)**.**

$$\mathit{unify}\ \kappa\ v_1\ v_2 \leqslant \kappa$$

Lemma 4.3.1.9 (unify_types)**.**

$$\left.\begin{array}{l} \varnothing; U(\kappa) \vdash v_1 : \overline{T} \\ \varnothing; U(\kappa) \vdash v_2 : \overline{T} \\ \vdash \kappa \end{array}\right\} \Rightarrow \vdash \mathit{unify}\ \kappa\ v_1\ v_2$$

4.3.2 Properties of the Narrowing Semantics

With the above specification of constraint sets, we can proceed to proving our main theorems for the narrowing semantics: decreasingness, soundness and completeness. The first theorem, *decreasingness* states that we never add new valuations to our constraint sets; our semantics can only refine the denotation of the input κ.

Theorem 4.3.2.1 (Decreasingness)**.**

$$e \dashv \kappa \Downarrow_q^t \kappa' \vDash v \ \Rightarrow\ \kappa' \leqslant \kappa$$

Proof: By induction on the derivation of narrowing, using the lemmas about ordering for fresh (Lemma 4.3.1.2), sample (Lemma 4.3.1.5) and unify (Lemma 4.3.1.8), followed by repeated applications of the transitivity of $\leqslant$. □

We will also need a form of big-step preservation for Core Luck: if a constraint set κ is well typed and an expression e has type T in $U(\kappa)$ and the empty context, then if we narrow $e \dashv \kappa$ to obtain $\kappa' \vDash v$, κ' will be well typed and v will also have the same type T in $U(\kappa')$.

Theorem 4.3.2.2 (Preservation)**.**

$$\left.\begin{array}{l} e \rightrightarrows \kappa \Downarrow_q^t \kappa' \vDash v \\ U(\kappa) \vdash e : T \\ \vdash \kappa \end{array}\right\} \Rightarrow \left\{\begin{array}{l} U(\kappa') \vdash v : T \\ \vdash \kappa' \end{array}\right.$$

Proof: The proof can be found in Appendix A.

Soundness and Completeness

Soundness and completeness can be visualized as follows:

$$\begin{array}{ccc} e_p & \xrightarrow{\Downarrow} & v_p \\ {\scriptstyle \sigma \in [\![\kappa]\!]}\uparrow & & \uparrow{\scriptstyle \sigma' \in [\![\kappa']\!]} \\ e \rightrightarrows \kappa & \xrightarrow{\Downarrow_q^t} & v \vDash \kappa' \end{array}$$

Given the bottom and right sides of the diagram, soundness guarantees that we can fill in the top and left. That is, any narrowing derivation $e \rightrightarrows \kappa \Downarrow_t^q \kappa' \vDash v$ directly corresponds to some derivation in the predicate semantics, with the additional assumption that all the unknowns in e are included in the domain of the input constraint set κ (which can be replaced by a stronger assumption that e is well typed in κ).

Theorem 4.3.2.3 (Soundness)**.**

$$\left.\begin{array}{l} e \rightrightarrows \kappa \Downarrow_t^q \kappa' \vDash v \\ \sigma'(v) = v_p \ \wedge \ \sigma' \in [\![\kappa']\!] \\ \forall u.\ u \in e \Rightarrow u \in dom(\kappa) \end{array}\right\} \Rightarrow \ \exists \sigma\ e_p. \left\{\begin{array}{l} \sigma'|_\sigma \equiv \sigma \\ \sigma \in [\![\kappa]\!] \\ \sigma(e) = e_p \\ e_p \Downarrow v_p \end{array}\right.$$

Completeness guarantees the opposite direction: given a predicate derivation $e_p \Downarrow v_p$ and a "factoring" of e_p into an expression e and a constraint set κ such that for some valuation $\sigma \in [\![\kappa]\!]$ substituting σ in e yields e_p, and under the assumption that everything is well typed, there is always a nonzero probability of obtaining some factoring of v_p as the result of a narrowing judgment.

Theorem 4.3.2.4 (Completeness)**.**

$$\left.\begin{array}{l} e_p \Downarrow v_p \\ \sigma(e) = e_p \\ \sigma \in [\![\kappa]\!] \wedge \vdash \kappa \\ \varnothing; U(\kappa) \vdash e : T \end{array}\right\} \Rightarrow \left\{\begin{array}{l} \exists v\ \kappa'\ \sigma'\ q\ t. \\ \left\{\begin{array}{l} \sigma'|_\sigma \equiv \sigma \ \ \wedge \ \ \sigma' \in [\![\kappa']\!] \\ \sigma'(v) = v_p \\ e \dashv \kappa \Downarrow_q^t \kappa' \vDash v \end{array}\right. \end{array}\right.$$

The proofs of both soundness and completeness can be found in Appendix A.

4.3.3 Properties of the Matching Semantics

Before we proceed to the theorems for the matching semantics, we need a specification for the *union* and *rename* operations.

Specification of union

$$\left.\begin{array}{l} U(\kappa_1)|_{U(\kappa_1)\cap U(\kappa_2)} = U(\kappa_2)|_{U(\kappa_1)\cap U(\kappa_2)} \\ union\ \kappa_1\ \kappa_2 = \kappa \end{array}\right\}$$

$$\Rightarrow$$

$$\left\{\begin{array}{l} U(\kappa) = U(\kappa_1) \cup U(\kappa_2) \\ [\![\kappa]\!] = [\![\kappa_1]\!] \cup [\![\kappa_2]\!] \end{array}\right.$$

To take the *union* of two constraint sets, their typing maps must obviously agree on any unknowns present in both. The denotation of the *union* of two constraint sets is then just the union of their corresponding denotations.

Similar lemmas concerning types and ordering can be proved about *union*.

Lemma 4.3.3.1 (union_ordered)**.**

$$\kappa' = union\ \kappa_1\ \kappa_2 \ \Rightarrow\ \kappa' \leqslant \kappa_1 \wedge \kappa' \leqslant \kappa_2$$

Lemma 4.3.3.2 (union_types)**.**

$$\left.\begin{array}{l} \kappa' = \mathit{union}\ \kappa_1\ \kappa_2 \\ \quad \vdash \kappa_1 \\ \quad \vdash \kappa_2 \end{array}\right\} \Rightarrow \vdash \kappa'$$

Specification of rename The *rename* function as introduced in the previous section can be encoded in terms of *fresh* and a function that renames a single unknown to the result of fresh, iteratively.

Properties The decreasingness property for the matching semantics is very similar to the narrowing semantics: if the matching semantics yields $\{\kappa'\}$, then κ' is smaller than the input constraint set.

Theorem 4.3.3.3 (Decreasingness)**.**

$$p \Leftarrow e \dashv \kappa \Uparrow_q^t \{\kappa'\} \ \Rightarrow\ \kappa' \leqslant \kappa$$

Proof: This is again the simplest proof: by induction on the derivation of matching judgment, using the lemmas about ordering for fresh (Lemma 4.3.1.2), sample (Lemma 4.3.1.5) and unify (Lemma 4.3.1.8) and repeated applications of the transitivity of $\leqslant$. □

Preservation is simpler than before since we only deal with a single output.

Theorem 4.3.3.4 (Preservation)**.**

$$\left.\begin{array}{l} p \Leftarrow e \dashv \kappa \Uparrow_q^t \{\kappa'\} \\ U(\kappa) \vdash e : \overline{T} \\ U(\kappa) \vdash p : \overline{T} \\ \vdash \kappa \end{array}\right\} \Rightarrow \vdash \kappa'$$

Soundness is again similar to the matching semantics.

Theorem 4.3.3.5 (Soundness)**.**

$$\left.\begin{array}{l} p \Leftarrow e \dashv \kappa \Uparrow_q^t \{\kappa'\} \\ \sigma'(p) = v_p \wedge \sigma' \in [\![\kappa']\!] \\ \forall u.\ (u \in e \vee u \in p) \Rightarrow u \in dom(\kappa) \end{array}\right\} \Rightarrow \exists \sigma\ e_p. \left\{\begin{array}{l} \sigma'|_\sigma \equiv \sigma \\ \sigma \in [\![\kappa]\!] \\ \sigma(e) = e_p \\ e_p \Downarrow v_p \end{array}\right.$$

For the completeness theorem, we need to slightly strengthen its premise; since the matching semantics may explore both branches of a *case*, it can fall into a loop when the predicate semantics would not (by exploring a non-terminating branch that the predicate semantics does not take). Thus, we require that *all* valuations in the input constraint set result in a terminating execution.

Theorem 4.3.3.6 (Completeness)**.**

$$\left.\begin{array}{l} e_p \Downarrow v_p \ \wedge\ \sigma \in [\![\kappa]\!] \\ \varnothing; U(\kappa) \vdash e : \overline{T} \ \wedge\ \vdash \kappa \\ \sigma(e) = e_p \wedge \sigma(p) = v_p \\ \forall \sigma' \in [\![\kappa]\!].\ \exists v'.\ \sigma'(e) \Downarrow v' \end{array}\right\} \Rightarrow \begin{array}{l} \exists \kappa'\ \sigma'\ q\ t. \\ \left\{\begin{array}{l} \sigma'|_\sigma \equiv \sigma \\ \sigma' \in [\![\kappa']\!] \\ p \Leftarrow e \dashv \kappa \Uparrow_q^t \{\kappa'\} \end{array}\right. \end{array}$$

Once again, the full proofs can be found in Appendix A.

4.4 Implementation

We next describe the Luck prototype: its top level, its treatment of backtracking, and the implementation of primitive integers instantiating the abstract specification presented in Section 4.3.1.

4.4.1 The Luck Top Level

The inputs provided to the Luck interpreter consist of an expression e of type *bool* (that is, 1+1), containing zero or more free unknowns $\vec{u}$ (but no free variables), and

an initial constraint set κ providing types and finite domains[3] for each unknown in $\vec{u}$, such that their occurrences in e are well typed $(\varnothing; U(\kappa) \vdash e : 1 + 1)$.

The interpreter matches e against `True` (that is, $L_{1+1}\ ()$), to derive a refined constraint set κ':

$$L_{1+1}\ () \Leftarrow e \dashv \kappa \Uparrow_q^t \{\kappa'\}$$

This involves random choices, and there is also the possibility that matching fails (and the semantics generates $\varnothing$ instead of $\{\kappa'\}$). In this case, a simple *global* backtracking approach could simply try the whole thing again (up to an ad hoc limit). While not strictly necessary for a correct implementation of the matching semantics, some *local* backtracking allows wrong choices to be reversed quickly and leads to an enormous improvement in performance [32]. Our prototype backtracks locally in calls to *choose*: if *choose* has two choices available and the first one fails when matching the instantiated expression against a pattern, then we immediately try the second choice instead. Effectively, this means that if e is already known to be of the form $L\_\ \_$, then narrow will not choose to instantiate it using $R\_\ \_$, and vice versa. This may require matching against e twice, and our implementation shares work between these two matches as far as possible. (It also seems useful to give the user explicit control over where backtracking occurs, but we leave this for future work.)

After the interpreter matches e against `True`, all the resulting valuations $\sigma \in [\![\kappa']\!]$ should map the unknowns in $\vec{u}$ to some values. However, there is no guarantee that the generator semantics will yield a κ' mapping every $\vec{u}$ to a unique values. The Luck top-level then applies the *sample* constraint set function to each unknown in $\vec{u}$, ensuring that $\sigma|_{\vec{u}}$ is the same for each σ in the final constraint set. The interpreter returns this common $\sigma|_{\vec{u}}$ if it exists, and backtracks otherwise.

[3] This restriction to finite domains appears to be crucial for our technical development to work, as discussed in the previous section. In practice, we have not yet encountered a situation where it was important to be able to generate examples of *unbounded* size (as opposed to examples up to some large maximum size). We do sometimes want to generate structures containing large numbers, since they can be represented efficiently, but here, too, choosing an enormous finite bound appears to be adequate for the applications we've tried. The implementation allows for representing all possible ranges of a corresponding type up to a given size bound. Such bounds are initialized at the top level, and they are propagated (and reduced a bit) to fresh unknowns created by pattern matching before these unknowns are used as inputs to the interpreter.

4.4.2 Pattern Match Compiler

In Section 4.1, we saw an example using a standard `Tree` datatype and instantiation expressions assigning different weights to each branch. While the desugaring of simple pattern matching to core Luck syntax is straightforward (4.2.1), nested patterns—as in Figure 4.17—complicate things in the presence of probabilities. We expand such expressions to a tree of simple case expressions that match only the outermost constructors of their scrutinees. However, there is generally no unique choice of weights in the expanded predicate: a branch from the source predicate may be duplicated in the result. We guarantee the intuitive property that the *sum* of the probabilities of the clones of a branch is proportional to the weights given by the user, but that still does not determine the individual probabilities that should be assigned to these clones.

The most obvious way to distribute weights is to simply share the weight equally with all duplicated branches. But the probability of a single branch then depends on the total number of expanded branches that come from the same source, which can be hard for users to determine and can vary widely even between sets of patterns that appear similar. Instead, Luck's default weighing strategy works as follows. For any branch B from the source, at any intermediate case expression of the expansion, the subprobability distribution over the immediate subtrees that contain at least one branch derived from B is uniform. This makes modifications of the source patterns in nested positions affect the distribution more locally.

In Figure 4.17, the `False` branch should have probability $\frac{1}{3}$. It is expanded into four branches, corresponding to subpatterns `Var _`, `Lam _ _`, `App (Var _) _`, `App (App _ _) _`. The latter two are grouped under the pattern `App _ _`, while the former two are in their own groups. These three groups receive equal shares of the total probability of the original branch, that is $\frac{1}{9}$ each. The two branches for `App (Var _) _` and `App (App _ _) _` split that further into twice $\frac{1}{18}$. On the other hand, `True` remains a single branch with probability $\frac{2}{3}$. The weights on the left of every pattern are calculated to reflect this distribution.

```
data T = Var Int | Lam Int T | App T T

sig isRedex :: T → Bool          -- Original
fun isRedex t =
  case t of
  | 2 % App (Lam _ _) _ → True -- 2/3
  | 1 % _ → False               -- 1/3

sig isRedex :: T → Bool          -- Expansion
fun isRedex t =
  case t of
  | 1 % Var _ → False           -- 1/9
  | 1 % Lam _ _ → False         -- 1/9
  | 7 % App t1 _ →
      case t1 of
      | 1 % Var _ → False       -- 1/18
      | 12 % Lam _ _ → True     -- 2/3
      | 1 % App _ _ → False     -- 1/18
```

Figure 4.17: Expanding case expression with a nested pattern and a wildcard. Comments show the probability of each alternative.

4.4.3 Constraint Set Implementation

Our desugaring of source-level pattern matching to core case expressions whose discriminee e is first narrowed means that rule **M-Case-1** is not executed for datatypes; only one of the evaluations of e against the L and R patterns will succeed and only one branch will be executed. This means that our constraint-set representation for datatypes doesn't need to implement *union*. We leverage this to provide a simple and efficient implementation of the unification constraints. For our prototype, the constraint solving behavior of *case* is only exploited in our treatment of primitive integers, which we detail at the end of this section.

The constraint set interface could be implemented in a variety of different ways. The simplest would be to explicitly represent constraint sets as sets of valuations, but this would lead to efficiency problems, since even unifying two unknowns would require traversing the whole set, filtering out all valuations in which the unknowns are different. On the other extreme, we could represent a constraint set as an arbitrary logical formula over unknowns. While this is a compact representation, it does not directly support the per-variable sampling that we require.

For our prototype we choose a middle way, using a simple data structure we call *orthogonal maps* to represent sets of valuations. An orthogonal map is a map from unknowns to *ranges*, which have the following syntax:

$$r ::= () \mid u \mid (r, r) \mid \mathit{fold}\ r \mid L\ r \mid R\ r \mid \{L\ r, R\ r\}$$

Ranges represent sets of non-functional values: units, unknowns, pairs of ranges, and L and R applied to ranges. We also include the option for a range to be a pair of an L applied to some range and an R applied to another. For example, the set of all Boolean values can be encoded compactly in a range (eliding folds and type information) as $\{L(), R()\}$. Similarly, the set $\{0, 2, 3\}$ can be encoded as $\{L(), R(R\{L(), R()\})\}$, assuming a standard Peano encoding of natural numbers.

However, while this compact representation can represent all sets of naturals, not all sets of Luck non-functional values can be precisely represented. For instance the set $\{(0, 1), (1, 0)\}$ cannot be represented using ranges, only approximated to $(\{L(), R(L())\},\ \{L(), R(L())\})$, which represents the larger set $\{(0, 0), (0, 1),\ (1,$

$0), (1, 1)\}$. This corresponds to a form of Cartesian abstraction, in which we lose any relation between the components of a pair, so if one used ranges as an abstract domain for abstract interpretation it would be hard to prove say sortedness of lists. Ranges are a rather imprecise abstract domain for algebraic datatypes [69, 72, 93].

We implement constraint sets as pairs of a typing environment and an optional map from unknowns to ranges. The typing environment of a constraint set ($U(\cdot)$ operation), is just the first projection of the tuple. A constraint set κ is SAT if the second element is not $\varnothing$. The *sample* primitive indexes into the map and collects all possible values for an unknown.

The only interesting operation with this representation is *unify*. It is implemented by straightforwardly translating the values to ranges and unifying those. For simplicity, unification of two ranges r_1 and r_2 in the presence of a constraint set κ returns both a constraint set κ' where r_1 and r_2 are unified and the unified range r'. If $r_1 = r_2 = ()$ there is nothing to be done. If both ranges have the same top-level constructor, we recursively unify the inner subranges. If one of the ranges, say r_1, is an unknown u we index into κ to find the range r_u corresponding to u, unify r_u with r_2 in κ to obtain a range r', and then map u to r' in the resulting constraint set κ'. If both ranges are unknowns u_1, u_2 we unify their corresponding ranges to obtain r'. We then pick one of the two unknowns, say u_1, to map to r', while mapping u_2 to u_1. To keep things deterministic we introduce an ordering on unknowns and always map u_i to u_j if $u_i < u_j$. Finally, if one range is the compound range $\{L\ r_{1l}, R\ r_{1r}\}$ while the other is $L\ r_2$, the resulting range is only L applied to the result of the unification of r_{1l} and r_2.

It is easy to see that if we start with a set of valuations that is representable as an orthogonal map, non-*union* operations will result in constraint sets whose denotation is still representable, which allows us to get away with this simple implementation of datatypes. The **M-Case-1** rule is used to model our treatement of integers. We introduce primitive integers in our prototype accompanied by standard integer equality and inequality constraints. In Section 4.2.5 we saw how a recursive less-than function can be encoded using Peano-style integers and *case* expressions that do *not* contain instantiation expressions in the discriminee. All integer constraints can be desugared into such recursive functions with the exact

same behavior—modulo efficiency.

To implement integer constraints, we extend the codomain of the mapping in the constraint set implementation described above to include a compact representation of sets of intervals of primitive integers as well as a set of the unknown's associated constraints. Every time the domain of an unknown u is refined, we use an incremental variant of the AC-3 *arc consistency* algorithm [88] to efficiently refine the domains of all the unknowns linked to u, first iterating through the constrains associated with u and then only through the constraints of other "affected" unknowns.

4.5 Evaluation

To evaluate the expressiveness and efficiency of Luck's hybrid approach to test case generation, we tested it with a number of small examples and two significant case studies: generating well-typed lambda terms and information-flow-control machine states. The Luck code is generally much smaller and cleaner than that of existing handwritten generators, though the Luck interpreter takes longer to generate each example—around 20× to 24× for the more complex generators. Finally, while this is admittedly a subjective impression, we found it significantly easier to get the generators right in Luck.

4.5.1 Small Examples

The literature on random test generation includes many small examples—list predicates such as `sorted`, `member`, and `distinct`, tree predicates like BSTs (Section 4.1) and red-black trees, and so on. In Appendix B, we show the implementation of many such examples in Luck, illustrating how we can write predicates and generators together with minimal effort.

We use red-black trees to compare the efficiency of our Luck interpreter to generators provided by commonly used tools like QuickCheck (random testing), SmallCheck (exhaustive testing) and Lazy SmallCheck [112]. Lazy SmallCheck leverages Haskell's laziness to greatly improve upon out-of-the-box QuickCheck and

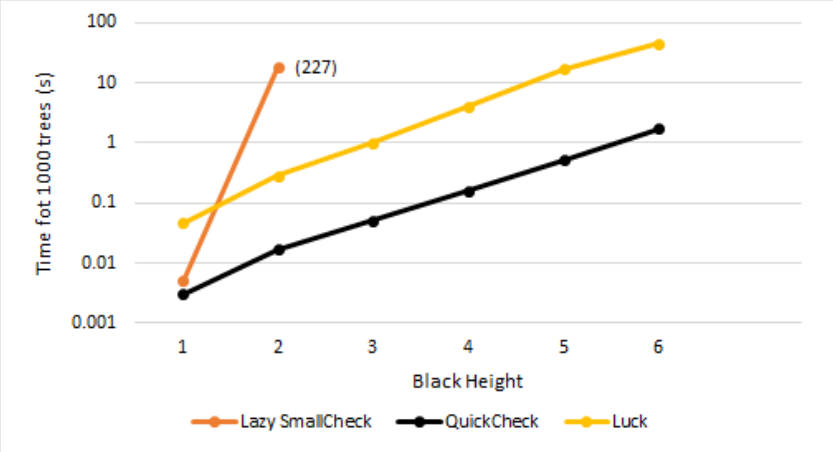


Figure 4.18: Red-Black Tree Experiment

SmallCheck generators in the presence of sparse preconditions, by using partially defined inputs to explore large parts of the search space at once. Using both Luck and Lazy SmallCheck, we attempted to generate 1000 red black trees with a specific black height bh—meaning that the depth of the tree can be as large as $2 \cdot bh + 1$. Results are shown in Fig. 4.18. Lazy SmallCheck was able to generate all 227 trees of black height 2 in 17 seconds, fully exploring all trees up to depth 5. When generating trees of black height 3, which required exploring trees up to depth 7, Lazy SmallCheck was unable to generate 1000 red black trees within 5 minutes. At the same time, the Luck implementation lies consistently within an order of magnitude of a very efficient handwritten QuickCheck generator that generates valid Red-Black trees directly. Using rejection-sampling approaches by generating trees and discarding those that don't satisfy the red-black tree invariant (e.g., QuickCheck or SmallCheck's `==>`) is prohibitively costly: these approaches perform much worse than Lazy SmallCheck.

4.5.2 Well-Typed Lambda Terms

Using our prototype implementation we reproduced the experiments of Pałka *et al.* [103], who generated well-typed lambda terms in order to discover bugs in GHC's strictness analyzer. We also use this case study to indirectly compare to

two narrowing-based tools that are arguably closer to Luck and that use the same case study to evaluate their work: Claessen*et al.* [31, 32] and Fetscher*et al.* [43].

We encoded a model of simply typed lambda calculus with polymorphism in Luck, providing a large typing environment with standard functions from the Haskell Prelude to generate interesting well-typed terms. The generated ASTs were then pretty-printed into Haskell syntax and each one was applied to a partial list of the form: `[1,2,undefined]`. Using the same version of GHC (6.12.1), we compiled each application twice: once with optimizations (`-O2`) and once without and compared the outputs.

A straightforward Luck implementation of a type system for the polymorphic lambda calculus was not adequate for finding bugs efficiently. To improve its performance we borrowed tricks from the similar case study of Fetscher *et al.* [43], seeding the environment with monomorphic versions of possible constants and increasing the frequency of `seq`, a basic Haskell function that introduces strictness, to increase the chances of exercising the strictness analyzer. Using this, we discovered bugs that seem similar (under quick manual inspection) to those found by Pałka *et al.* and Fetscher *et al.*.

Luck's generation speed was slower than that of Pałka's handwritten generator. We were able to generate terms of average size 50 (internal nodes), and, grouping terms together in batches of 100, we got a total time of generation, unparsing, compilation and execution of around 35 seconds per batch. This is a slowdown of 20x compared to that of Pałka's. However, our implementation is a total of 82 lines of fairly simple code, while the handwritten development is 1684 lines, with the warning "...the code is difficult to understand, so reading it is not recommended" in its distribution page [101].

The derived generators of Claessen *et al.* [31] achieved a 7x slowdown compared to the handwritten generator, while the Redex generators [43] also report a 7x slowdown in generation time for their best generator. However, by seeding the environment with monomorphised versions of the most common constants present in the counterexamples, they were able to achieve a time per counterexample on par with the handwritten generator.

4.5.3 Information-Flow Control

For a second large case study, we turned to the information-flow control case study of Chapter 3, re-implementing methods for generating indistinguishable machine states. Given an abstract stack machine with data and instruction memories, a stack, and a program counter, one attaches *labels*—security levels—to runtime values, propagating them during execution and restricting potential flows of information from *high* (secret) to *low* (public) data. The desired security property, *termination-insensitive noninterference*, states that if we start with two indistinguishable abstract machines `s1` and `s2` (i.e., all their low-tagged parts are identical) and run each of them to completion, then the resulting states `s1'` and `s2'` are also indistinguishable.

In "Testing Noninterference, Quickly" [65], we found that efficient testing of this property could be achieved in two ways: either by generating instruction memories that allow for long executions and checking for indistinguishability at each low step (called *LLNI*, low-lockstep noninterference), or by looking for counter-examples to a stronger invariant (strong enough to *prove* noninterference), generating two arbitrary indistinguishable states and then running for a single step (*SSNI*, single step noninterference). In both cases, there is some effort involved in generating indistinguishable machines: for efficiency, one must first generate one abstract machine `s` and then *vary* `s`, to generate an indistinguishable one `s'`. In writing such a generator for variations, one must effectively reverse the indistinguishability predicate between states and then keep the two artifacts in sync.

We first investigated the stronger property (SSNI), by encoding the indistinguishability predicate in Luck and using our prototype to generate small, indistinguishable pairs of states. In 216 lines of code we were able to describe both the predicate and the generator for indistinguishable machines. The same functionality required >1000 lines of complex Haskell code in the handwritten version. The handwritten generator is reported to generate an average of 18400 tests per second, while the Luck prototype generates an average of 1450 tests per second, around 12.5 times slower.

The real promise of Luck, however, became apparent when we turned to LLNI.

In Chapter 3, to generate long sequences of instructions we used *generation by execution*: starting from a machine state where data memories and stacks are instantiated, they generate the current instruction ensuring it does not cause the machine to crash, then allow the machine to take a step and repeat. While intuitively simple, this extra piece of generator functionality took significant effort to code, debug, and optimize for effectiveness, resulting in more than 100 additional lines of code. The same effect was achieved in Luck by the following 6 intuitive lines, where we just put the previous explanation in code:

```
sig runsLong :: Int -> AS -> Bool
fun runsLong len st =
    if len <= 0 then True
    else case step st of
        | 99 % Just st' -> runsLong (len - 1) st'
        | 1 % Nothing  -> True
```

We evaluated our generator on the same set of buggy information-flow analyses. We were able to find all of the same bugs, with similar effectiveness (number of bugs found per 100 tests). However, the Luck generator was 24 times slower (Luck: 150 tests/s, Haskell: 3600 tests/s). We expect to be able to improve this result (and the rest of the results in this section) with a more efficient implementation that compiles Luck programs to QuickCheck generators directly, instead of interpreting them in a minimally tuned prototype.

The success of the prototype in giving the user enough flexibility to achieve similar effectiveness with state-of-the-art generators, while significantly reducing the amount of code and effort required, suggests that the approach Luck takes is promising and points towards the need for a real, optimizing implementation.

Acknowledgments

The work presented in this Chapter was the basis for the POPL 2017 paper "Beginner's Luck" [81], with Diane Gallois-Wong, Catalin Hriţcu, John Hughes, Benjamin Pierce and Li-yao Xia. While the majority of the work presented in this section

is mine, with the exception of the pattern match expansion algorithm that is attributed to Li-yao, this work would not have been possible without the constant discussions about the semantics of Luck with all the collaborators and especially Benjamin.

Chapter 5

Generating Good Generators for Inductive Relations

In Chapter 2, we introduced QuickChick, a property-based testing QuickCheck clone for Coq and demonstrated its functionality. In particular, compared to similar tools in other proof assistants like Isabelle [17], QuickChick gives the user the full customizability that QuickCheck provides: one can easily write and compose generators using an established combinator library.

However, as we saw earlier, for complex properties and especially specifications involving sparse preconditions, setting up PBRT-style testing can involve substantial work. Writing generators for well-distributed random data for such properties can be both complex and time consuming, sometimes to the point of being a research contribution in its own right [65, 66, 103]!

In the previous chapter, we identified two techniques for automatically deriving a generator from a given precondition: narrowing and constraint solving. Automatic narrowing-based generators can achieve testing effectiveness (measured as bugs found per test generated) comparable to hand-written custom generators, even for challenging examples [31, 43, 81].

Unfortunately, both hand-written and narrowing-based automatic generators are subject to bugs. For hand-written ones, this is because generators for complex conditions can often also be complex, often more than the condition itself;

moreover, they must be kept in sync if the condition is changed, another source of errors. Automatic generators do not suffer from the latter problem, but narrowing solvers are themselves rather complex beasts, whose correctness is therefore questionable. Even Luck, which we presented in Chapter 4, that comes with a proof of correctness, only proves an abstract model of the core algorithm, not the rather large Haskell implementation.

Bugs in generators can come in two forms: they can generate too much, or too little—i.e., they can be either unsound or incomplete. Unsoundness can lead to false positives, which can waste significant amounts of time. Incompleteness can lead to ineffective testing, where certain bugs in the program under test can never be found because the generator will never produce an input that provokes them. Both problems can be detected—unsoundness by double-checking whether generated values satisfy the property, incompleteness by techniques such as mutation testing [71]—and unsoundness can be mitigated by filtering away generated values that fail the double-check, but incompleteness bugs can require substantial effort to understand and repair.

The core contribution of this Chapter is a method for compiling a large class of logical conditions, expressed as Coq inductive relations, into random generators together with soundness and completeness proofs for these generators. We do *not* prove that the compiler itself is correct in the sense that it can only produce good generators; rather, we adopt a *translation validation* approach [107] where we produce a checkable certificate of correctness along with each generator. A side benefit of this approach is that, by compiling inductive relations into generators, we avoid the interpretive overhead of existing narrowing-based generators. As discussed in the previous Chapter, this overhead is one of the reasons existing generators can be an order of magnitude slower than their hand-written counterparts.

We have implemented our method as an extension of QuickChick. Using QuickChick, a Coq user can write down desired properties like

```
Conjecture preservation : forall (t t' : tm) (T : ty),
  |- t \in T  ->  t ===> t'  ->  |- t' \in T.
```

and look for counterexamples with no additional effort:

```
QuickChick preservation. ⟶ QuickChecking preservation...
Passed 10000 tests
```

The technical contributions of this chapter are as follows:

- We present a Luck-inspired method for compiling a large class of inductive definitions into random generators. Section 5.1 introduces our compilation algorithm through a sequence of progressively more complex examples; Section 5.2 describes it in full detail.

- We show how this algorithm can also be used to produce proofs of (possibilistic) correctness for every derived generator (Section 5.3). Indeed, by judicious application of Coq's typeclass features, we can use exactly the same code to produce both generators and proof terms.

- To evaluate the applicability of our method, we applied the QuickChick implementation to a large part of *Software Foundations* [106], a machine-checked textbook on programming language theory. Of the 232 nontrivial theorems we considered, 84% are directly amenable to PBRT (the rest are higher-order properties that would at least require significant creativity to validate by random testing); of these, 83% can be tested using our algorithm. We discuss these findings in detail in Section 5.4.1.

- To evaluate the efficiency of our generators, we compare them to fine-tuned handwritten generators for information-flow control abstract machines (Section 5.4.2) and for well typed STLC terms (Section 5.4.3). The derived generators were 1.75× slower than the custom ones, demonstrating a significant speedup over previous interpreted approaches such as Luck [81].

We conclude and draw directions for future work in Section 5.5. The implementation of the algorithm in QuickChick, further integrating testing and proving in the Coq proof assistant and providing more push-button-style automation while retaining customizability, is described in the next chapter (Section 6.1).

5.1 Good Generators, by Example

The main focus of this chapter is to derive *correct* generators for simply-typed data satisfying dependently-typed, inductive invariants. This section uses examples to showcase different behaviors that our generation algorithm needs to exhibit; the algorithm itself will be described more formally in the following section. In particular, we are going to give a few progressively more complex inductive characterizations of trees, and detail how we can automatically produce a generator for trees satisfying those characterizations. We first encountered Coq trees in Chapter 2. We repeat their standard definition here for the reader's convenience:

```
Inductive Tree A :=
| Leaf : Tree A
| Node : A -> Tree A -> Tree A -> Tree A.
```

5.1.1 Nonempty Trees

Our first example is nonempty trees, i.e., trees that are not just leaves.

```
Inductive nonempty : Tree → Prop :=
| NonEmpty : ∀ x l r, nonempty (Node x l r).
```

From a user's perspective, we can quickly come up with a generator for nonempty trees: we just need to create arbitrary `x`, `l` and `r` and combine them into a `Node`.

```
Definition gen_nonempty : G (option Tree) :=
  do x ← arbitrary;
  do l ← arbitrary;
  do r ← arbitrary;
  ret (Some (Node x l r)).
```

But how could we automate this process?

We know that we want to generate a tree `t` satisfying `nonempty`; that means that we need to pick some constructor of `nonempty` to satisfy. Since there is only one constructor, we only have one option, `NonEmpty`. By looking at the conclusion of the `NonEmpty` constructor we know that `t` must be a `Node`. This can

be described by a *unification* procedure. Specifically, we introduce an *unknown variable* `t` (similar to logical variables in logic programming, or unification variables in type inference) plus one unknown variable for each universally quantified variable of the constructor (here `x`, `l` and `r`). We then proceed to unify `t` with `(Node x l r)`. Since there are no more constraints—we call them "hypotheses"—in `NonEmpty`, and since `x`, `l` and `r` are still completely unknown, we instantiate them arbitrarily (using the `Gen` instance for natural numbers that is provided by default, as well as the instance for `Tree`s that can be derived automatically).

5.1.2 Complete Trees

For our second example of a condition, consider complete trees (also known as perfect trees): binary trees whose leaves are all at the same depth. The shape of a complete tree can be fully characterized by its depth: a complete tree of depth zero is necessarily a `Leaf`, while a complete tree of depth `n+1` is formed by combining two complete trees of depth `n` into a `Node`. This is reflected in the following inductive definition:

```
Inductive complete : nat → Tree → Prop :=
| CompleteLeaf :
    complete 0 Leaf
| CompleteNode : ∀ n x l r,
    complete n l → complete n r →
    complete (S n) (Node x l r).
```

Since `complete` has two parameters, we need to decide whether the derived generator produces all of them or treats some of them as inputs, i.e., we need to assign *modes* to the parameters, in the sense of functional logic programming. Let's assume that that first parameter is an input to the generator (called `in1`), and we want to generate trees `t` that satisfy `complete in1 t`. Once again, we introduce an unknown variable `t` (that we want to generate), as well as an unknown variable for `in1`: since the generator will receives `in1` as an argument, we don't know its actual value at derivation time!

We now have two constructors to choose from to try to satisfy, `CompleteLeaf`

and CompleteNode. If we pick CompleteLeaf we need to unify t with Leaf and in1 with O. Since t is unconstrained at this point, we can always unify it with Leaf. By contrast since we don't know the value of in1 at derivation time, we need to produce a *pattern match* on it: if in1 is O, then we can proceed to return a Leaf, otherwise we can't satisfy CompleteLeaf.

On the other hand, if we pick CompleteNode, we introduce new unknowns n, x, l and r for the universally quantified variables. We proceed to unify t with Node x l r and m with S n. Like before, we need to pattern match on in1 to decide at runtime if it is nonzero; we bind n in the pattern match and treat it as an input from that point onward. We then handle the recursive constraints on l and r, instantiating both the left and right subtrees with a recursive call to the generator we're currently deriving. Finally, x remains unconstrained so we instantiate it arbitrarily, like in the nonempty tree case.

```
Fixpoint gen_complete (in1 : nat) : G (option Tree) :=
  match in1 with
    | O => ret (Some Leaf)
    | S n => l ← gen_complete n ;;
             r ← gen_complete n ;;
             x ← arbitrary ;;
             ret (Some (Node x l r))
  end.
```

The complete inductive predicate is particularly well-behaved. First of all, for every possible input depth m there exists some tree t that satisfies complete m t. That will not necessarily hold in the general case. Consider for example an inductive definition that consists of only the CompleteLeaf constructor:

```
Inductive half_complete : nat → Tree → Prop :=
| CompleteLeaf' : half_complete 0 Leaf.
```

Once again, we will need to pattern match on m, and, if m is zero, we can proceed as in the previous definition of complete to return a Leaf. However, if m is nonzero there is nothing we can possibly do to return a valid tree that satisfies half_complete. This is the reason why our generators return options of the

underlying type:

```
Definition gen_half_complete (in1 : nat) : G (option Tree) :=
  match in1 with
    | O => ret (Some Leaf)
    | _ => ret None
  end.
```

Secondly, the usage of the input parameter serves as a structurally decreasing parameter for our fixpoint. In the general case that is not necessarily true and we will need to introduce a `size` parameter (like we did in the previous section for simple inductive types), as we will see in the next example.

5.1.3 Binary Search Trees

For a more complex example, consider binary search trees: for every node, each label in its left subtree is smaller than the node label while each label in the right subtree is larger. In Coq code, we could characterize binary search trees whose elements are between two extremal values `lo` and `hi` with the following code:

```
Inductive bst : nat → nat → Tree → Prop :=
| BstLeaf : ∀ lo hi,
    bst lo hi Leaf
| BstNode : ∀ lo hi x l r,
    lo < x → x < hi →
    bst lo x l → bst x hi r →
    bst lo hi (Node x l r).
```

A `Leaf` is always such a search tree since it contains no elements; a `Node` is such a search tree if its label `x` is between `lo` and `hi` and its left and right subtrees are appropriate search trees as well.

The derived generator (tweaked a bit for readability) is as follows; we explain it below:

```
Definition gen_bst in1 in2 : nat → G (option Tree) :=
  let fix aux_arb size (in1 in2 : nat) : G (option (Tree)) :=
```

```
    match size with
    | O => ret (Some Leaf)
    | S size' =>
      backtrack [ (1, ret (Some Leaf))
                ; (1, x ← arbitraryST (fun x => in1 < x) ;;
                      if (x < in2)? then
                        l ← aux_arb size' in1 x ;;
                        r ← aux_arb size' x in2 ;;
                        ret (Some (Node x l r))
                      else ret None
                )]
    end
in fun size => aux_arb size in1 in2.
```

This generator is bounded: just like in the previous section, we use a natural number `size` to serve as a limit in the depth of the derivation tree. When `size` is 0 we are only allowed to use constructors that do not contain recursive calls to the inductive type we're generating. In the binary search tree example, that means that we can only choose the `BstLeaf` constructor. In that case, we introduce unknowns `in1` and `in2` that correspond to the inputs to the generation, `t` that corresponds to the generated tree, as well as two unknowns `lo` and `hi` corresponding to the universally quantified variables of the `BstLeaf` case. We then try to unify `in1` with `lo`, `in2` with `hi`, and `t` with `Leaf`. Since `lo`, `hi` and `t` are unconstrained, the unification succeeds and our derived generator returns `Some Leaf`.

When `size` is not zero, we have a choice. We can once again choose to satisfy the `BstLeaf` constructor, which results in the generator returning `Some Leaf`. We can also choose to try to satisfy the recursive `BstNode` constructor. After introducing unknowns and performing the necessary unifications, we know that the end product of this sub-generator will be `Some (Node x l r)`. We then proceed to process the constraints that are enforced by the constructor.

To begin with, we encounter `lo < x`. Since `lo` is mapped to the input `in1`, we need to generate `x` such that `x` is (strictly) greater than `in1`. We do that by invoking the typeclass method `arbitraryST` for generating arbitrary natural

numbers satisfying the corresponding predicate. Now, when we encounter the `x < hi` constraint both `x` and `hi` are instantiated so we need to *check* whether or not the constraint holds. The notation `p?` looks for a `Dec` instance of `p` to serve as the boolean condition for the `if` statement. If it does, we proceed to satisfy the rest of the constraints by recursively calling our generator. If not, we can no longer produce a valid binary search tree so we must fail, returning `None`.

One additional detail in the generator is the use of the `backtrack` combinator instead of `frequency` to choose between different constructor options. The `backtrack` combinator operates exactly like `frequency` to make the first choice—choosing a generator with type `G (option A)` based on the induced discrete distribution. However, should the chosen generator fail, it backtracks and chooses a different generator until it either exhausts all options or the backtracking limit.

5.1.4 Nonlinearity

As a last example, we will use an artificial characterization of "good" trees to showcase one last difficulty that arises in the context of dependent inductive types: *non-linear patterns*.

```
Inductive goodTree : nat → nat → Tree → Prop :=
| GoodLeaf : ∀ n, goodTree n n Leaf.
```

In this example, `goodTree in1 in2 t` only holds if the tree `t` is a `Leaf` and `in1` and `in2` are equal, as shown by the non-linear occurrence of `n` in the conclusion of `GoodLeaf`. If we assume that both `in1` and `in2` will be inputs to our generator, then this will translate to an equality check in the actual generator.

```
Fixpoint gen_good (in1 in2 : nat) size : G (option Tree) :=
  match size with
  | 0 => backtrack [(1, if in1 = in2 ? then ret (Some Leaf)
                        else ret None)]
  | S _ => backtrack [(1, if in1 = in2 ? then ret (Some Leaf)
                          else ret None)]
  end.
```

We can see the equality check `in1 = in2 ?` in the derived generator above. We can also see that the structure of the generator is similar to the one for binary search trees, even though it seems unnecessary. In particular, we encounter calls to `backtrack` with a single element list (which is equivalent to just the inner generator), as well as an unnecessary match on the `size` parameter with duplicated branch code. This uniform treatment of generators facilitates the proof term generation of Section 5.3. In addition, we could obtain the simpler and slightly more efficient generators by a straight-forward optimization pass.

5.2 Generating Good Generators

We now describe the generalized narrowing algorithm more formally.

5.2.1 Input

Our generation procedure targets simply-typed inductive data, which satisfy a particular form of dependently-typed inductive relations. More precisely, we take as input an inductively defined relation R with p arguments of types $A_1, A_2, \cdots, A_p$, where each A_i is a simple inductive type. Each constructor C in the definition of R takes as arguments some number of universally quantified variables ($\overline{x}$) and some preconditions—each consisting of an inductive predicate S applied to constructor expressions (only consisting of constructors and variables) $\overline{e}$; its conclusion is R

itself applied to constructor expressions $e_1, e_2, \cdots, e_p$.

$$\begin{array}{c}\texttt{Inductive}\ R : A_1 \to A_2 \to \cdots \to A_p \to \texttt{Prop} \quad := \\ \ldots \quad | \quad C : \forall\ \overline{x},\ \overline{S\ \overline{e}} \to R\ e_1\ e_2\ \cdots\ e_p \quad | \quad \ldots\end{array}$$

In Section 5.4 we demonstrate the applicability of this class in practical situations, in and discuss possible extensions to this format as future work (Section 5.5).

5.2.2 Unknowns and Ranges

We first need to formalize *unknowns*, which are used to keep track of sets of potential values that variables can take during generation, just like in Luck. One important difference is that sometimes unknowns will be provided as *inputs* to the generation algorithm; this means that they can only take a single fixed value, but that value is not known at derivation time. Looking back at the `complete` trees example, we knew that `in1` would be an input to `gen_complete`. However, when deriving the generator we could not make any assumptions about `in1`: we could not freely unify it with `O` for instance—we had to pattern match against it.

We represent sets of potential values as *ranges*.

$$r := \mathit{undef}_\tau \mid \mathit{fixed} \mid u \mid C\ \overline{r}$$

The first option for the range of an unknown is *undefined* (parameterized by a type). The unknowns we want to generate (such as `tree`, in the binary search tree example) start out with undefined ranges. On the other hand, a range can also be *fixed*, signifying that the corresponding unknown's value serves as an input at runtime (`in1` and `in2` in the binary search tree example). Next, a range of an unknown can also be a different unknown, to facilitate sharing. Finally, a range can be a constructor C fully applied to a list of ranges.

We use a map from unknowns to ranges, written κ, to track knowledge about unknowns during generation. For each constructor C, we initialize this map with the unknowns that we want to generate mapped to undef_τ appropriate types τ, the rest of the parameters to R mapped to *fixed*, and the universally quantified

variables of C also mapped to appropriate undefined ranges. For instance, to generate a `tree` such that `bst in1 in2 tree` holds for all `in1` and `in2`, the initial map for the `BstNode` constructor would contain `in1` and `in2` mapped to *fixed*, `tree` mapped to $undef_{\texttt{Tree}}$, and the unknowns `lo`, `hi`, `x`, `l` and `r` introduced by `BstNode` mapped to corresponding undefined ranges:

$$\begin{aligned}\kappa \;:=\; & (\texttt{in1} \mapsto \mathit{fixed}) \oplus (\texttt{in2} \mapsto \mathit{fixed}) \\ \oplus\; & (\texttt{tree} \mapsto \mathit{undef}_{\texttt{Tree}}) \\ \oplus\; & (\texttt{lo} \mapsto \mathit{undef}_{\texttt{nat}}) \oplus (\texttt{hi} \mapsto \mathit{undef}_{\texttt{nat}}) \\ \oplus\; & (\texttt{x} \mapsto \mathit{undef}_{\texttt{Nat}}) \oplus (\texttt{l} \mapsto \mathit{undef}_{\texttt{Tree}}) \oplus (\texttt{r} \mapsto \mathit{undef}_{\texttt{Tree}})\end{aligned}$$

5.2.3 Overview

We have already hinted at the general structure of the generation algorithm in Section 5.1. Let's assume `in1` ... `inn` will be the inputs to the generator and that `out1` ... `outm` will be the outputs. We then produce a bounded generator that takes `in1` through `inn` as inputs, as well as an additional natural number parameter `size`:

```
Fixpoint aux_arb size in1 ... inn :=
  match size with
  | O => backtrack [ ... (w_C, g_C) ...]
  | S size' => backtrack [ ... (w_C, g_C) ...]
  end.
```

Both when `size` is zero and when it is not, we use `backtrack` to choose between a number of generators. In the latter case, we have one sub-generator g_C for each constructor C. The former case is nearly the same, except that the sub-generators that perform recursive calls to `aux_arb` are filtered out of the list. The weights to backtrack (w_c) can be chosen by the user via lightweight annotations, similar to the local distribution control of Luck, as we will see in the evaluation section (5.4.2). The general structure of each g_C appears in Figure 5.1.

The outer component of every sub-generator will be a sequence of pattern

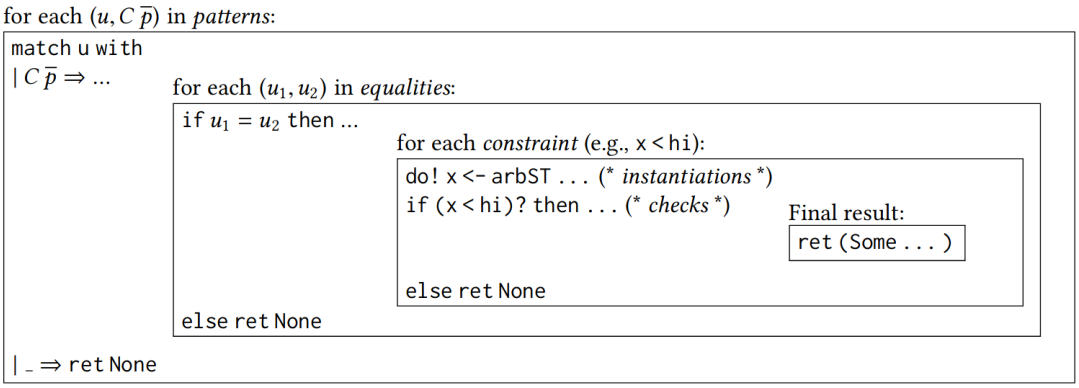


Figure 5.1: General Structure of each Sub-generator

matches: unification will sometimes signify that we need to match an unknown against a pattern. For instance, in the case of `complete` trees we needed to match `in1` against `O`. Each such pattern match has two branches: one that is considered successful and allows generation to continue; and one that catches all other possible cases and fails (returns `None`).

After nesting all possible matches, we need to ensure that any equalities raised by the unification hold. In the successful branch of the innermost match (if any), we start a sequence of if-equality-statements. For example, in the case of `good` trees that were demonstrating non-linear patterns, we checked that `in1 = in2` before continuing with the generation.

The equalities are followed by a sequence of instantiations and checks that are enforced by the hypotheses of C. Looking back at the binary search tree example, we needed to generate a random `x` such that `x` was greater than the lower bound `in1`; we also needed to check whether that generated `x` was less than the upper bound `in2`.

Finally, we combine all the unknowns that we wanted to generate for in a `Some` to return them as the final result. Note that, just like in the `nonEmpty` trees example, we might need to perform a few more instantiations if some unknowns necessary remain completely unconstrained.

$$\text{type } \mathit{Pattern} = \mathit{Unknown} \mid C\ \overline{\mathit{Pattern}}$$

$$\text{record } \mathit{UnifyState} = \left\{ \begin{array}{lcl} \mathit{constraints} & : & \mathit{Map\ Unknown\ Range} \\ \mathit{equalities} & : & \mathit{Set\ (Unknown, Unknown)} \\ \mathit{patterns} & : & \mathit{List\ (Unknown, Pattern)} \\ \mathit{unknowns} & : & \mathit{Set\ Unknown} \end{array} \right\}$$

$$\text{type } \mathit{Unify\ a} = \mathit{UnifyState} \to \mathit{option\ (a, UnifyState)}$$

$$\begin{array}{l} \mathit{update} : \mathit{Unknown} \to \mathit{Range} \to \mathit{Unify}\ () \\ \mathit{update}\ u\ r = \lambda s. \\ \quad \text{let } \kappa = \mathit{constraints}(s) \text{ in} \\ \quad \mathit{Some}\ ((),\ s\{\mathit{constraints} \leftarrow \kappa[u \mapsto r]\}) \end{array}$$

$$\begin{array}{l} \mathit{equality} : \mathit{Unknown} \to \mathit{Unknown} \to \mathit{Unify}\ () \\ \mathit{equality}\ u_1\ u_2 = \lambda s. \\ \quad \text{let } \mathit{eqs} = \mathit{equalities}(s) \text{ in} \\ \quad \mathit{Some}\ ((),\ s\{\mathit{equalities} \leftarrow \{u_1 = u_2\} \bigcup \mathit{eqs}\}) \end{array}$$

$$\begin{array}{l} \mathit{pattern} : \mathit{Unknown} \to \mathit{Pattern} \to \mathit{Unify}\ () \\ \mathit{pattern}\ u\ p = \lambda s. \\ \quad \text{let } \mathit{ps} = \mathit{patterns}(s) \text{ in} \\ \quad \mathit{Some}\ ((),\ s\{\mathit{patterns} \leftarrow (u, p) :: \mathit{ps}\}) \end{array}$$

$$\begin{array}{l} \mathit{fresh} : \mathit{Unify\ Unknown} \\ \mathit{fresh} = \lambda s. \\ \quad \text{let } \mathit{us} = \mathit{unknowns}(s) \text{ in} \\ \quad \text{let } u = \mathit{fresh\_unknown}\ (\mathit{us}) \text{ in} \\ \quad \mathit{Some}\ (u,\ s\{\mathit{unknowns} \leftarrow u \bigcup \mathit{us}\}) \end{array}$$

Figure 5.2: Unification Monad

5.2.4 Unification

The most important component of the derivation algorithm is the unification. For every constructor C with conclusion $R\ e_1\ e_2\ \cdots\ e_p$, we convert each e_i to a range and unify it with the corresponding unknown argument of R. For instance, in the binary search tree example, we would unify the `in1`, `in2`, and `tree` unknowns with `lo`, `hi`, and `Node x l r` respectively.

The entire unification algorithm is written inside a state-option monad, presented in Figure 5.2. To keep track of information about unknowns we use a *Map* from *Unknown*s to *Range*s; to track necessary equalities—like in the `good` tree of the previous section—we keep a *Set* of pairs of unknowns; to produce the necessary pattern matches—like in `complete` trees—we gather them in a *List*; finally, to be able to produce fresh unknowns on demand, we keep all existing unknowns in a *Set*.

Each of the four components of the state can be modified through specific monadic actions. The *update* action sets the range of an unknown; the *equality* action registers a new equality check; *pattern* adds a pattern match; and *fresh* gen-

$$
\begin{array}{lcl}
\mathit{unify}\ u_1\ u_2 & = & \begin{cases} \mathit{return}\ () & \text{if } u_1 = u_2 \\ r_1 \leftarrow \kappa[u_1];\ r_2 \leftarrow \kappa[u_2];\ \mathit{unifyR}\ (u_1, r_1)\ (u_2, r_2) & \text{otherwise} \end{cases} \\
\mathit{unify}\ (C_1\ r_{11}\ \cdots\ r_{1n})\ (C_2\ r_{21}\ \cdots\ r_{2m}) & = & \mathit{unifyC}\ (C_1\ r_{11}\ \cdots\ r_{1n})\ (C_2\ r_{21}\ \cdots\ r_{2m}) \\
\mathit{unify}\ u_1\ (C_2\ r_{21}\ \cdots\ r_{2m}) & = & r_1 \leftarrow \kappa[u_1];\ \mathit{unifyRC}\ (u_1, r_1)\ (C_2\ r_{21}\ \cdots\ r_{2m}) \\
\mathit{unify}\ (C_1\ r_{11}\ \cdots\ r_{1n})\ u_2 & = & r_2 \leftarrow \kappa[u_2];\ \mathit{unifyRC}\ (u_2, r_2)\ (C_1\ r_{11}\ \cdots\ r_{1n}) \\
\\
\mathit{unifyR}\ (u_1, \mathit{undef}_\tau)\ (u_2, r) & = & \mathit{update}\ u_1\ u_2 \\
\mathit{unifyR}\ (u_1, r)\ (u_2, \mathit{undef}_\tau) & = & \mathit{update}\ u_2\ u_1 \\
\mathit{unifyR}\ (u_1, u_1')\ (u_2, r) & = & \mathit{unify}\ u_1'\ u_2 \\
\mathit{unifyR}\ (u_1, r)\ (u_2, u_2') & = & \mathit{unify}\ u_1\ u_2' \\
\mathit{unifyR}\ (\_, C_1\ r_{11}\ \cdots\ r_{1n})\ (\_, C_2\ r_{21}\ \cdots\ r_{2m}) & = & \mathit{unifyC}\ (C_1\ r_{11}\ \cdots\ r_{1n})\ (C_2\ r_{21}\ \cdots\ r_{2m}) \\
\mathit{unifyR}\ (u_1, \mathit{fixed})\ (u_2, \mathit{fixed}) & = & \mathit{equality}\ u_1\ u_2;\ \mathit{update}\ u_1\ u_2 \\
\mathit{unifyR}\ (u_1, \mathit{fixed})\ (\_, C_2\ r_{21}\ \cdots\ r_{2m}) & = & \mathit{match}\ u_1\ (C_2\ r_{21}\ \cdots\ r_{2m}) \\
\mathit{unifyR}\ (\_, C_1\ r_{11}\ \cdots\ r_{1n})\ (u_2, \mathit{fixed}) & = & \mathit{match}\ u_2\ (C_1\ r_{11}\ \cdots\ r_{1n}) \\
\\
\mathit{unifyC}\ (C_1\ r_{11}\ \cdots\ r_{1n})\ (C_2\ r_{21}\ \cdots\ r_{2m}) & = & \begin{cases} \mathit{fold}\ \mathit{unify}\ \overline{(r_{1i}, r_{2i})} & \text{if } C_1 = C_2 \text{ and } n = m \\ \bot & \text{otherwise} \end{cases} \\
\\
\mathit{unifyRC}\ (u, \mathit{undef}_\tau)\ (C_2\ r_{21}\ \cdots\ r_{2m}) & = & \mathit{update}\ u_1\ (C_2\ r_{21}\ \cdots\ r_{2m}) \\
\mathit{unifyRC}\ (u, u')\ (C_2\ r_{21}\ \cdots\ r_{2m}) & = & r \leftarrow \kappa[u'];\ \mathit{unifyRC}\ (u', r)\ (C_2\ r_{21}\ \cdots\ r_{2m}) \\
\mathit{unifyRC}\ (u, \mathit{fixed})\ (C_2\ r_{21}\ \cdots\ r_{2m}) & = & \mathit{match}\ u\ (C_2\ r_{21}\ \cdots\ r_{2m}) \\
\mathit{unifyRC}\ (u, C_1\ r_{11}\ \cdots\ r_{1n})\ (C_2\ r_{21}\ \cdots\ r_{2m}) & = & \mathit{unifyC}\ (C_1\ r_{11}\ \cdots\ r_{1n})\ (C_2\ r_{21}\ \cdots\ r_{2m}) \\
\\
\mathit{match}\ u\ (C\ r_1\ \cdots\ r_n) & = & \overline{p} \leftarrow \mathit{mapM}\ \mathit{matchAux}\ \overline{r};\ \mathit{pattern}\ u\ (C\ \overline{p}) \\
\\
\mathit{matchAux}\ (C\ \overline{r}) & = & \overline{p} \leftarrow \mathit{mapM}\ \mathit{matchAux}\ \overline{r};\ \mathit{return}\ (C\ \overline{p}) \\
\mathit{matchAux}\ u & = & r \leftarrow \kappa[u];\ \mathit{case}\ r\ \mathit{of} \\
 & & \quad \mathit{undef}_\tau \Rightarrow \mathit{update}\ u\ \mathit{fixed} \\
 & & \mid \mathit{fixed} \Rightarrow u' \leftarrow \mathit{fresh};\ \mathit{equality}\ u'\ u;\ \mathit{update}\ u'\ u;\ \mathit{return}\ u' \\
 & & \mid u' \Rightarrow \mathit{matchAux}\ u' \\
 & & \mid C\ \overline{r} \Rightarrow \overline{p} \leftarrow \mathit{mapM}\ \mathit{matchAux}\ \overline{r};\ \mathit{return}\ (C\ \overline{p})
\end{array}
$$

Figure 5.3: Unification Algorithm

erates and returns a new unknown. We write $\kappa[u]$ for the action that looks up an unknown, and we write ; for the monadic bind operation and $\bot$ to signify failure (the constant action $\lambda s.\ \mathit{Nothing}$).

The main unification procedure, *unify*, is shown in Figure 5.3. At the top level, we only need to consider three cases for unification—unknown-unknown, constructor-constructor, and unknown-constructor—because the e_1 through e_p are constructor expressions containing only constructors and variables, which are translated to constructor ranges and unknowns respectively. Most cases are unsurprising; the main important difference from regular unification is the need to handle potentially fixed—but not statically known—inputs.

Case $u_i \mapsto \mathit{undef}$**:** If the range of either of the unknowns, say u_1, is undefined, we update κ so that u_1 points to u_2 instead. From that point on, they correspond to exactly the same set of potential values. Consider the `goodTree` example of the previous section, where in the initial map for `GoodLeaf` we have unknowns `in1` and `in2` as inputs to the generator, `tree` as the unknown being generated, and `n` introduced by `GoodLeaf`:

$$\kappa := (\texttt{in1} \mapsto \mathit{fixed}) \oplus (\texttt{in2} \mapsto \mathit{fixed}) \oplus (\texttt{tree} \mapsto \mathit{undef}_{\texttt{Tree}}) \oplus (\texttt{n} \mapsto \mathit{undef}_{\texttt{nat}})$$

We first unify `in1` with `n`; since $\texttt{n} \mapsto \mathit{undef}_{\texttt{nat}}$ in the initial map, the unification updates that map such that $\texttt{n} \mapsto \texttt{in1}$.

Case $u_i \mapsto u_i'$**:** If either unknown maps to another unknown we recursively try to unify using the new unknown as input. For example, when we try to unify `in2` with `n` in the updated map for `GoodLeaf`, we recurse and attempt to unify `in1` with `in2`.

Case $u_1 \mapsto C_1\ r_{11}\ \cdots\ r_{1n}$ **and** $u_2 \mapsto C_2\ r_{21}\ \cdots\ r_{2m}$**:** If both ranges have some constructor at their head, there are two possibilities: either $C_1 \neq C_2$, in which case the unification fails, or $C_1 = C_2$ and $n = m$, in which case we recursively unify r_{1i} with r_{2i} for all i. We maintain the invariant that all the ranges that appear as arguments to any constructor contain only constructors and unknowns, which allows us to call *unify* and reduce the total number of cases.

The last two cases, dealing with *fixed* ranges, are the most interesting ones.

Case $u_1 \mapsto \mathit{fixed}$ **and** $u_2 \mapsto \mathit{fixed}$**:** If both u_1 and u_2 map to a *fixed* range in κ, then we need to assert that whatever the values of u_1 and u_2 are, they are equal. This will translate to an equality check between u_1 and u_2 in the derived generator. We record this necessary check using *equality* and proceed assuming that the check succeeds, setting one unknown's range to the other. Continuing with the `goodTree` example, when we attempt to unify `in1` and `in2`, both have *fixed* ranges. This results in the equality check $n_1 = n_2$ that appears in `gen_good`.

Case $u_i \mapsto$ *fixed* and $u_j \mapsto C\ r_1\ \cdots\ r_n$: The last possible configuration pairs a *fixed* range against a constructor range $C\ r_1\ \cdots\ r_n$. This will result in a pattern match in the derived generator. We saw such an example in the previous section in the form of `complete'`. One branch of the match will be against a representation of the range $C\ r_1\ \cdots\ r_n$ and lead to success, while the other branch will terminate the generation with failure in all other cases. To match against $C\ r_1\ \cdots\ r_n$, we will need to convert all of the ranges $\overline{r}$ to patterns $\overline{p}$, while dealing with potentially non-linear appearances of unknowns inside the constructor range. This is done by traversing the ranges $\overline{r}$, applying a helper function *matchAux* to each, and logging the result in the state monad using *pattern*.

If r is itself a constructor C, we need to recursively traverse its ranges, convert them to patterns $\overline{p}$ and combine them into a single pattern $C\ \overline{p}$. If r is an unknown u, we look up its range inside the current map. If it is undefined we can use u as the bound variable in the pattern; we update the binding of u in the map to be *fixed*, as it will be extracting information out of the fixed discriminee. On the other hand, if the range is *fixed*, we need to create a *fresh* unknown u', use that as the pattern variable and then enforce an equality check between u and u'. Finally, the unknown and constructor cases result in appropriate recursions.

5.2.5 Handling Hypotheses

Another important part of the derivation of a generator for a single constructor C is handling all of its hypotheses. Given a hypothesis of the form $S\ e_1\ e_2\ \cdots\ e_m$, we once again identify a few different cases.

If there is exactly one undefined variable amongst the e_i, we need to instantiate it. That translates either to a call to the generic `arbitraryST` function, or to a recursive call to the currently derived generator. The bst predicate provides examples of both: after the unification is complete, the map κ will have the following form:

$$\begin{aligned}\kappa \ :=\ & (\texttt{in1} \mapsto \textit{fixed}) \oplus (\texttt{in2} \mapsto \textit{fixed}) \oplus (\texttt{tree} \mapsto \texttt{Node x l r}) \\ \oplus\ & (\texttt{lo} \mapsto \texttt{in1}) \oplus (\texttt{hi} \mapsto \texttt{in2}) \\ \oplus\ & (\texttt{x} \mapsto \textit{undef}_{\texttt{Nat}}) \oplus (\texttt{l} \mapsto \textit{undef}_{\texttt{Tree}}) \oplus (\texttt{r} \mapsto \textit{undef}_{\texttt{Tree}})\end{aligned}$$

When processing the hypothesis `lo < x`, the unknown `lo` maps to `in1`which in

turn is *fixed*, while `x` is still undefined. Thus, to generate `x` such that `lo < x` holds, we need to invoke the `arbitraryST` method of `GenSuchThat` for `(fun x => in1 < x)`. After processing this constraint, the range of `x` becomes to *fixed*: we know that it has a concrete value but not what it is. For all intents and purposes it can be treated as if it was an input to the generation from this point on.

$$\begin{aligned}\kappa \;&:=\; (\texttt{in1} \mapsto \textit{fixed}) \oplus (\texttt{in2} \mapsto \textit{fixed}) \oplus (\texttt{tree} \mapsto \texttt{Node x l r}) \\ &\oplus\; (\texttt{lo} \mapsto \texttt{in1}) \oplus (\texttt{hi} \mapsto \texttt{in2}) \oplus (\texttt{x} \mapsto \textit{fixed}) \oplus (\texttt{l} \mapsto \textit{undef}_{\texttt{Tree}}) \oplus (\texttt{r} \mapsto \textit{undef}_{\texttt{Tree}})\end{aligned}$$

Therefore, when processing the `bst lo x l`, only `l` is unconstrained. However, since generating `l` such that `bst lo x l` holds is exactly the generation mode we are currently deriving, we just make a recursive call to `aux_arb` to generate `l`.

The second possibility for a hypothesis is that all expressions e_i are completely fixed, in which case we can only check whether this hypothesis holds. For example, when we encounter the `x < hi` constraint, both `x` and `hi` have already been instantiated and therefore we need to check whether `x < hi` holds at runtime, using the `dec` method of the decidability typeclass.

A final possibility is that a hypothesis could contain multiple undefined unknowns. Deciding which of them to instantiate first and how many at a time is a matter of heuristics. For example, if in the constraint `bst lo hi t`, if all of `lo`, `hi` and `t` were undefined, we could pick to make a call to `arbitraryST bst`, or we could instantiate arguments one at a time. In our implementation, we prioritize recursive calls whenever possible; we leave further exploration and comparison of different heuristics as future work.

5.2.6 Assembling the Final Result

After processing all hypotheses we have an updated constraint map κ, where, compared to the constraint map after the unification, some unknowns have had their ranges *fixed* as a result of instantiation. However, there might still be remaining unknowns that are undefined. Such was the case for the `nonEmpty` tree example where `x`, `l` and `r` were all still undefined. Thus, we must iterate through κ, instantiating any unknowns u for which $\kappa[u] = \textit{undef}$. To complete the generator g_C

for a particular constructor, we look up the range of all unknowns that are being generated, convert them to a Coq expression, group them in a tuple and return them.

5.2.7 Putting it All Together

A formal presentation of the derivation for a single constructor is shown in Figure 5.4. Here, for simplicity of exposition, we allow only a single output *out*. In general, even though our implementation of the algorithm deals with a single output as well, the algorithm presented in this section can handle an arbitrary number of outputs.

Given an inductive relation R and a particular constructor $C : \forall\ \overline{x},\ \overline{S\ \overline{e}} \rightarrow P\ e_1\ e_2\ \cdots\ e_p$, our goal is to generate *out* such that for all $\overline{in}$, the predicate $R\ e'_1\ e'_2\ \ldots\ e'_p$ holds via constructor C, where the e's are constructor expressions containing only variables in $\{out\} \bigcup \overline{in}$. First, we create an initial map κ as described in Subsection 5.2.2. We use it to construct an initial state *st* for the unification monad (Subsection 5.2.4), where the *patterns* and *equalities* fields are empty, while the *unknowns* field holds $\overline{in}$, *out* and all universally quantified variables of C. We then evaluate a sequence of monadic actions, each one attempting to unify e_i with its corresponding e'_i. If at any point the unification fails, the constructor C is not inhabitable and we fail. If it succeeds, we proceed to produce all of the nested pattern matches and equalities in order (*emit_patterns* and *emit_equalities*), as described in Subsection 5.2.3. Afterwards, we process all the hypotheses using *emit_hypotheses* as described in Subsection 5.2.5, emitting instantiations or checks as appropriate, while updating the constraint set κ. Finally, we complete the generation by instantiating all unknowns that are still undefined and constructing the result by reading off the range of *out* in the final constraint set (5.2.7).

5.3 Generating Correctness Proofs

This section describes how we automatically generate proofs that our generators are sound and complete with respect to the inductive predicates they were derived

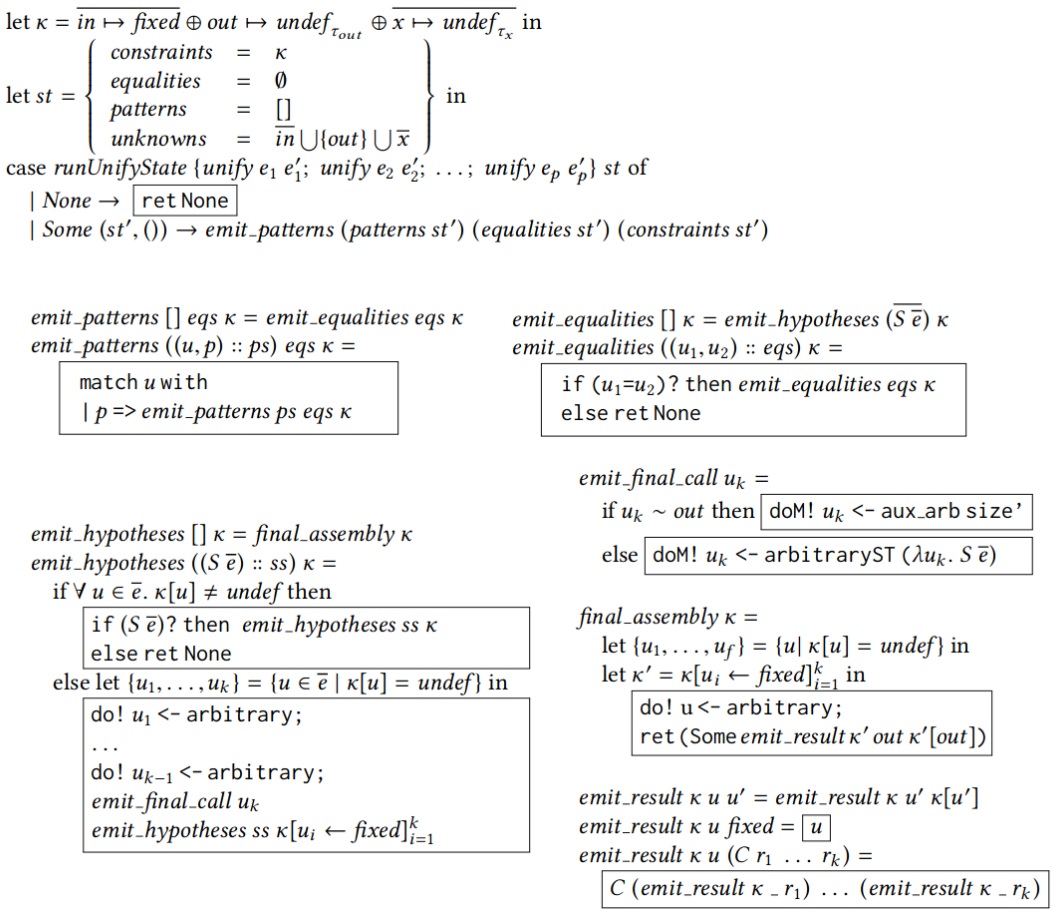


Figure 5.4: Derivation of one case of a generator g_C (for a single constructor C), in pseudo-code. Boxes delimit "quasi-quoted" Coq generator code to be emitted. Inside boxes, *italic* text indicates "anti-quoted" pseudo-code whose result is to be substituted in its place.

from. Following the translation validation approach of Pnueli *et al.* [107], rather than proving once and for all that *every* generator we build is guaranteed to be correct, we build a proof term certifying that each specific generator is correct at the same time as we build the generator itself. In fact, the same algorithm that is used to compile generators from inductive predicates is also used to compile their corresponding proofs of correctness. We leverage an existing verification framework for QuickChick, designed to allow users to (manually) prove soundness and completeness of generators built from QuickChick's low-level primitives [105].

This verification framework assigns semantics to each generator by mapping it to its set of outcomes, i.e. the elements that have non-zero probability of being generated. This enables proving that all the elements in a set of outcomes satisfy some desired predicate (soundness), and that all the elements that satisfy the predicate are in the set of outcomes (completeness). To ease reasoning about user-defined generators, QuickChick provides a library of lemmas that specify the behavior of built-in combinators.

We leverage this framework to specify the set of outcomes of derived generators: given an inductive relation and some input parameters, it should be exactly the set of elements that satisfy the inductive relation. Automatic generation of proofs is analogous to generation of generators and is done using the same algorithm. Just as generators are derived by composing generator combinators that we select by examining the structure of the inductive predicate, proofs are derived by composing the corresponding correctness lemmas that are provided by QuickChick. We glue these proof components together in order to obtain proofs for unsized generators using typeclass resolution, just as we did to obtained unsized generators. To enable this, we extend the typeclass infrastructure of QuickChick to encode properties of generators as typeclasses and we automatically generate instances of these classes for the derived generators.

Section 5.3.1 briefly describes QuickChick's verification framework, focusing on the proof generation machinery. Section 5.3.2 outlines the structure of the generated proofs and describes all the terms, definitions, and proofs that we need to generate in order to obtain the top-level correctness proof. Finally, Section 5.3.3 describes the extensions to the typeclass infrastructure of QuickChick that we made

in order to facilitate proof generation.

5.3.1 Verification Framework

QuickChick assigns semantics to generators by mapping them to the set of values that have non-zero probability of being generated. Recall from Chapter 2 that generators are functions mapping a random seed and a natural number to an element of the underlying type. The semantics of a generator for a given size parameter is exactly the values that can be generated for this particular size parameter.

$$[\![g]\!]_s = \{\ x \mid \exists r,\ g\ s\ r = x\ \}$$

We can then define the semantics of a generator by taking the union of these sets over all possible size parameters.

$$[\![g]\!] = \bigcup_{s \in \mathbb{N}} [\![g]\!]_s$$

It may seem as though we could have skipped the first definition and inlined its right-hand side in the second. However, by separating out the first definition we can additionally characterize the behavior of generators with respect to the size parameter. For instance, we can define the class of size-monotonic generators, whose set of outcomes for a given size parameter is included to the set of outcomes for every larger size parameter.

$$\texttt{sizeMonotonic}\ g \stackrel{\text{def}}{=} \forall s_1\ s_2,\ s_1 \leqslant s_2 \rightarrow [\![g]\!]_{s_1} \subseteq [\![g]\!]_{s_2}$$

Another useful class of generators is bound-monotonic generators, i.e., bounded generators that behave monotonically with respect to their bound parameter. Recall that bounded generators are parameterized by a natural number which bounds the size of the generated terms.

$$\texttt{boundMonotonic}\ g \stackrel{\text{def}}{=} \forall s\ s_1\ s_2,\ s_1 \leqslant s_2 \rightarrow [\![g\ s_1]\!]_s \subseteq [\![g\ s_2]\!]_s$$

Together, these characterizations allow us to obtain convenient specifications for combinators. To support reasoning about size-monotonicity properties, we encode them as typeclasses and provide lemmas (encoded as typeclass instances) that various generator combinators are size monotonic if all the involved generators are size monotonic. For instance, here is the lemma for monadic binding:

$$\frac{\texttt{sizeMonotonic}\ g \qquad \forall x \in [\![g]\!],\ \texttt{sizeMonotonic}\ (f\ x)}{\texttt{sizeMonotonic}\ (g >\!\!>\!\!= f)}\ \textsc{monBind}$$

There is a similar lemma that guarantees that `sized` is size monotonic but it requires both bound and size monotonicity for the bounded generator.

$$\frac{\texttt{sizeMonotonic}\ g \qquad \forall s,\ \texttt{boundMonotonic}\ (g\ s)}{\texttt{sizeMonotonic}\ (\texttt{sized}\ g)}\ \textsc{monSized}$$

To prove that `sized` is monotonic, we also need a second premise that requires the bounded generator to be bound monotonic. This is because `sized` will use the internal size parameter of `G` to instantiate the bound parameter of the generator.

To support reasoning about generators, QuickChick provides a library of lemmas that specify the semantics of generator combinators and can be used to compositionally verify user defined generators. These lemmas can be seen as a proof theory for the `G` monad; one can apply them in order to build derivations that computations in this monad are correct.

The simplest example of a correctness lemma is the one of `ret`. Unsurprisingly, the semantics of the return of the `G` monad is just a singleton set.

$$\frac{}{[\![\texttt{ret}\ x]\!] = \{\ x\ \}}\ \textsc{semRet}$$

The lemma for monadic bind is more interesting. In particular, the expected specification that composes the set of outcomes of the two generators using an indexed union is true, but under the additional requirement that the generators

involved are size monotonic.

$$\frac{\texttt{sizeMonotonic}\ g \qquad \forall x \in [\![g]\!],\ \texttt{sizeMonotonic}\ (f\ x) \qquad [\![g]\!] = s \qquad \forall x \in s,\ [\![f\ x]\!] = h\ x}{[\![g >>= f]\!] = \bigcup_{x \in s} h\ x}\ \textsc{semBind}$$

The intuition behind this requirement [105] is that the set on the left-hand side of semBind contains elements that are generated when the same size parameter is threaded to both generators, whereas the right-hand side indexes over elements that have been generated by g when the size parameter ranges over all natural numbers. To address this mismatch, we use monotonicity. In particular, to obtain the right to left inclusion we can pick a witness for the size parameter that is greater than both of the size parameters we obtain as witnesses from the hypothesis, such as their sum (or max) and then use monotonicity to prove the inclusion.

The correctness lemma for `sized` is crucial for proof generation, as it gives us proofs about unbounded generators. It states that the semantics of the combinator is the union of the sets of outcomes of the bounded generator indexed over all natural numbers.

$$\frac{\texttt{boundMonotonic}\ g \qquad (\forall x \in \mathbb{N},\ \texttt{sizeMonotonic}\ (g\ x)) \qquad \forall x \in \mathbb{N},\ [\![g\ x]\!] = f\ x}{[\![\texttt{sized}\ g]\!] = \bigcup_{x \in \mathbb{N}} f\ x}\ \textsc{semSized}$$

The lemma requires that the bounded generator is size monotonic for all bounds and, in addition, that it is monotonic in the bound parameter itself. These conditions are required because the set on the left-hand size of the specification contains elements that are generated from g using same number for the size and the bound, whereas in the right-hand side the bound and the size parameter range independently over natural numbers. As in the case of the monadic bind, we can work around this mismatch using monotonicity.

5.3.2 Proof Generation

This section describes the proof terms that we generate for each derived generator. To describe the structure of the constructed proof terms, we will use the generator for binary search trees presented in 5.1 as a running example. The terms themselves have the same structure as the generators and are generated using using the same algorithms, replacing the generator building blocks with their proof counterparts. For brevity, we assume a single output of the generation procedure; we can easily encode multiple outputs using tuples.

Top-level proof Let $R : A \to Prop$ be an inductive predicate and $g : \texttt{option}\ A$ a derived generator for this predicate. We want to generate proofs that g is sound and complete with respect to this predicate, i.e., that the set of outcomes of the generator is exactly the elements that satisfy P:

$$\texttt{isSome} \cap [\![g]\!] = \texttt{Some}[P]$$

Since our generators can fail (their return type is `option` A), we need to take the image of P under `Some`. We also remove `None` from the set of outcomes of g by intersecting it with `isSome`, i.e., the set of elements whose outermost constructor is `Some`.

In the case of binary search trees, this amounts to saying that the set of outcomes of the unbounded generator (which we obtain using `sized`; it is automatically derived by typeclass resolution) is exactly the set of trees that satisfy the `bst` predicate for some given inputs.

$$\forall\ in_1\ in_2,\ \texttt{isSome} \cap [\![\texttt{sized}\ (\texttt{gen\_bst}\ in_1\ in_2)]\!] = \texttt{Some}[\texttt{bst}\ in_1\ in_2]$$

As expected, to generate proofs about unbounded generators we have to first generate proofs about bounded generators. These proofs can be then lifted using the specification of `sized` that we saw in the previous section.

Proofs for Bounded Generators Before deriving correctness proofs for such bounded generators, we first need to settle on a specification. To this end, for each inductive definition for which we derive a generator, we generate automatically an operator, which we call `iter`, that maps a natural number to the set of elements that inhabit the inductive relation and whose proof has height less or equal to the given natural number. This set will serve as a specification for the bounded generator for a given size parameter.

This operator has exactly the same shape as the generator, and it is obtained using the same algorithm; the only thing that changes is that, instead of using the combinators of the `G` (`option` $-$) monad, we use those of the `set` monad. For instance, in the case of the `bst` predicate the `iter` operator looks like the following:

$$\texttt{iter\_bst}\ in_1\ in_2\ s = \begin{cases} \text{if } s = 0, \\ \quad \{\ \texttt{Leaf}\ \}, \\ \text{if } s = s' + 1, \\ \quad \{\ \texttt{Leaf}\ \} \cup \\ \quad \bigcup_{x > in_1} \texttt{if } x < in_2 \texttt{ then} \\ \qquad \bigcup_{l \in (\texttt{iter\_bst}\ in_1\ x\ s')} \bigcup_{r \in (\texttt{iter\_bst}\ x\ in_2\ s')} \{\ \texttt{Node}\ x\ l\ r\ \} \\ \quad \texttt{else } \varnothing \end{cases}$$

The parallels with the generator stand out: we can obtain this by replacing `ret` (`Some` $-$) with the singleton set (i.e., the return of the `set` monad),n `bind` with indexed union (i.e., the bind of the `set` monad), and `ret None` with empty set (i.e., the fail action of the `set` monad).

Using `iter` we can accurately characterize the set of outcomes of a bounded generator:

$$\texttt{isSome} \cap [\![g\ n]\!] = \texttt{Some}[\texttt{iter}\ n]$$

The proof term for this proposition also has the same structure as the generator, but this time, instead of monadic combinators, we use the corresponding proof rules. Since we only care to specify the `Some` part of the set of outcomes of the generators, we can use slightly modified proof rules that require a weaker notion

of generator monotonicity. In particular, the new rules only require the generator to be monotonic in the `Some` part of its set of outcomes. This is captured by the following two definitions.

$$\texttt{sizeMonotonicOpt}\ g \stackrel{\text{def}}{=} \forall s_1\ s_2,\ s_1 \leqslant s_2 \rightarrow \texttt{isSome} \cap [\![g]\!]_{s_1} \subseteq \texttt{isSome} \cap [\![g]\!]_{s_2}$$

$$\texttt{boundMonotonicOpt}\ g \stackrel{\text{def}}{=} \forall s\ s_1\ s_2,\ s_1 \leqslant s_2 \rightarrow \texttt{isSome} \cap [\![g\ s_1]\!]_s \subseteq \texttt{isSome} \cap [\![g\ s_2]\!]_s$$

Using the above definitions we can formulate the alternative proof rules. Below are examples of reformulated lemmas for `bind` and `size`.

$$\frac{\texttt{sizeMonotonicOpt}\ g \qquad \forall x,\ \texttt{sizeMonotonicOpt}\ (f\ x) \qquad [\![g]\!] = s \qquad \forall x,\ \texttt{isSome} \cap [\![f\ x]\!] = h\ x}{\texttt{isSome} \cap [\![g >\!\!>\!\!= f]\!] = \bigcup_{x \in s} h\ x}\ \textsc{semOptBind}$$

$$\frac{\texttt{boundMonotonicOpt}\ g \qquad \forall n \in \mathbb{N},\ \texttt{sizeMonotonicOpt}\ (g\ n)) \qquad \forall n,\ \texttt{isSome} \cap [\![g\ n]\!] = h\ n}{\texttt{isSome} \cap [\![\texttt{sized}\ g]\!] = \bigcup_{n \in \mathbb{N}} h\ n}\ \textsc{semOptSized}$$

Our goal is to lift specification from bounded to unbounded generators using the corresponding lemma for `sized`. To that end, we need a proof that the union of these sets produced by `iter` over all natural numbers is exactly the set of elements that satisfy the predicate.

$$\bigcup_{n \in \mathbb{N}} \texttt{iter}\ n = P$$

The above proof also requires us to generate a proof that these sets operators are monotonic in the size parameter.

$$\forall n_1\ n_2,\ n_1 \leqslant n_2 \rightarrow \texttt{iter}\ n_1 \subseteq \texttt{iter}\ n_2$$

Monotonicity proofs As described above, to produce correctness proofs we need to also produce monotonicity proofs for the unbounded generators. These proofs are used in both constructing the correctness proofs for bounded combina-

tors, as well as lifting them to unbounded ones. In order to be able to use the generators as individual components in other derived generators that come with correctness proofs, we also lift size monotonicity proofs to unbounded generators. As in previous cases, this is done using the corresponding lemma for `sized`. Again, we automate this process by providing the appropriate typeclass instances. Note that the choice to generate proofs of this weaker notion of monotonicity is not essential; we could have generated proofs of full monotonicity instead. However, we opted for this weaker notion as it significantly simplifies the proof of bound monotonicity.

5.3.3 Typeclasses for Proof Generation

As we did for generators in Section 5.2, we rely on typeclasses to connect individual proof components and lift specifications to unbounded generators. In this subsection we describe the extensions to the typeclass infrastructure of QuickChick presented in Chapter 2, which are needed in order to achieve this. We use Coq notation so that we can display the actual typeclass definitions; the notation `:&:` denotes set intersection and the notation `@:` the image of a function over some set.

Monotonicity First we extend the typeclass hierarchy to encode size and bound monotonicity properties.

```
Class SizeMonotonicOpt {A} (g : G (option A)) :=
  { monotonic_opt :
      ∀ s1 s2,
        s1 <= s2 →
        isSome :&: semGenSize g s1
        \subset
        isSome :&: semGenSize g s2 }.
```

```
Class BoundMonotonicOpt {A} (g : nat → G (option A)) :=
  { sizeMonotonicOpt :
      ∀ s s1 s2,
        s1 <= s2 →
        isSome :&: semGenSize (g s1) s
        \subset
        isSome :&: semGenSize (g s2) s }.
```

We automatically generate proofs that derived bounded generators are bound and size monotonic by explicitly constructing the proof term and we automatically create instances of these classes. Size monotonicity can then be derived for unbounded generators using the following provided instance.

```
Instance SizeMonotonicOptOfBounded (A : Type) (P : A → Prop)
         (H1 : GenSizedSuchThat A P)
         (H2 : ∀ s : nat, SizeMonotonicOpt (arbitrarySizeST P s))
         (H3 : BoundMonotonicOpt (arbitrarySizeST P))
: SizeMonotonicOpt (arbitraryST P).
```

Given a `GenSizedSuchThat` instance for a predicate P (`H1` above), which provides access to a constrained bounded generator `arbitrarySizeST P`, and instances of size and bound monotonicity for this generator (`H2` and `H3`), we can obtain an instance of size monotonicity for unbounded generator for this predicate, `arbitraryST P`, which is also obtained automatically by the corresponding instance.

Set Operators To express the correctness property of generators we introduce a typeclass that gives a generic interface to predicates which are equipped with an `iter` operator.

```
Class Iter {A : Type} (P : A → Prop) :=
  { iter : nat → set A;
    iter_mon : ∀ n1 n2, n1 <= n2 → iter n1 \subset iter n2;
    iter_spec : \bigcup_(n : nat) (iter n) ⟷ P }.
```

Correctness We can define a subclass of the above class, which is used to characterize bounded generators that are correct with respect to a predicate.

```
Class BoundedSuchThatCorrect {A} (P : A → Prop) {Iter A P}
      (g : nat → G (option A)) :=
  { boundedCorrect :
      ∀ s, isSome :&: semGen (g s) ←→ Some @: (iter s)
  }.
```

In the above, we are requiring that P is an instance of the `Iter` class in order to be able to use `iter` to express the correctness property. Following our usual practice, we also define a class for correct unbounded generators.

```
Class SuchThatCorrect {A} (P : A → Prop) (g : G (option A)) :=
  { correct :
      isSome :&: semGen g ←→ Some @: P
  }.
```

As before, we automatically generate instances for correctness of bounded generators by proving the proof terms, and we then lift them to unbounded generators by adding the corresponding instance.

```
Instance SuchThatCorrectOfBounded (A : Type) (P : A → Prop)
         (H1 : GenSizedSuchThat A P) (H2 : Iter P)
         (H3 : ∀ s : nat, SizeMonotonicOpt (arbitrarySizeST P s))
         (H4 : BoundMonotonicOpt (arbitrarySizeST P))
         (H5 : SizedSuchThatCorrect P (arbitrarySizeST P))
: SuchThatCorrect P (arbitraryST P).
```

The above instance is similar to the one for monotonicity but it additionally requires an instance for correctness of the unbounded generator (`H5`). It also requires an instance of the `Iter` class for `P` (`H2`). This instance is required as an (implicit) argument to the instance of correctness and also in the proof itself as it provides the specification of `iter`.

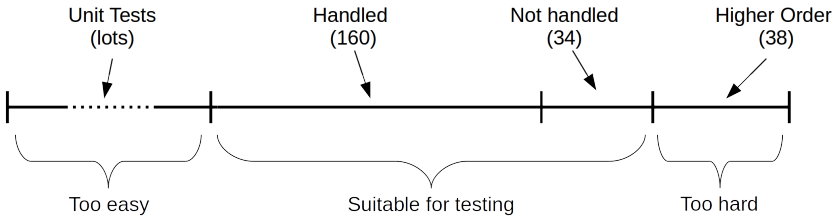


Figure 5.5: Evaluation Results

5.4 Evaluation

We evaluate two aspects of our generators: the applicability of the restricted class of inductive types we target (Section 5.4.1) and the efficiency of the derived generators compared to handwritten ones (Sections 5.4.2 and 5.4.3).

5.4.1 QuickChecking Software Foundations

To evaluate the applicability of our algorithm we tried to automatically test a large body of specifications that are representative of those commonly used in verifying properties of programming languages. Such a body of specifications can be found in Software Foundations [106], a machine-checked textbook for programming language theory and verification using Coq. We attempted to automatically test every theorem or lemma in the suggested main course of the textbook, all the way through the simply typed lambda calculus chapters.

Our findings are summarized in Figure 5.5. To avoid skewing our findings, we separately count certain classes of examples. Of the 232 nontrivial (non-unit-test) theorems we considered, 194 (84%) are directly amenable to PBRT; the 38 remaining theorems deal with generation for higher-order properties, which we deem too difficult for automatic test-case generation (we give examples below). Of the 194 theorems we believed "should be testable," 160 (83%) could be tested using our implemented algorithm. This demonstrates that the class of inductive propositions targeted by our narrowing generalization is broad enough to tackle many practical cases in the *Software Foundations* setting. The rest of this section discusses the different classes of theorems we considered and our methodology for testing each one.

First of all, *Software Foundations* incorporates a large number of unit tests, like the following test for disjunction:

```
Example test_orb1:  (orb true false) = true.
```

Such examples are trivially checkable and uninteresting from a generation perspective.

On the other hand, a class of lemmas that are completely out of scope of generation techniques deal with universally quantified higher-order properties. Consider the following canonical example of a Hoare triple:

```
Theorem hoare_seq : forall P Q R c1 c2,
     {{Q}} c2 {{R}}  ->  {{P}} c1 {{Q}}  ->  {{P}} c1;;c2 {{R}}.
```

Testing such a property would require generating constrained random elements of type `state -> Prop`, which is beyond current automatic random generation techniques. For context, the number of higher-order properties we excluded were 38; 36 of them came from the Logic and Hoare logic chapters that heavily use quantification over `Prop`s.

Finally, a third class of properties that could be interesting from a generation perspective but are a poor fit for property-based random testing are existential properties. For example, consider progress for a type system:

```
Conjecture progress :
  forall t T,  |- t \in T  ->  value t \/ exists t', t ===> t'.
```

While generating `t` and `T` such that `t` has type `T` is both interesting and possible within the extension of QuickChick presented in this paper, it is not possible to decide whether the conclusion of the property holds! However, most of the time, it is possible to rewrite existential conclusions into decidable ones. For example, for a deterministic step relation, we could write a partial `step` function and rewrite the conclusion to check whether the `t` can take a step: `isSome (step t)`.

With the above in mind we proceeded to automatically derive generators for all simple inductive types, generators for different modes for inductive relations, as well as proofs for both. We completely elided unit tests, counted (but otherwise ignored) properties that required generation of higher order properties, and

converted conclusions to decidable when necessary. We then turned each property into a `Conjecture`—an automatically admitted property—and attempted to test it with QuickChick. For example, the preservation property became:

```
Conjecture preservation : forall t t' T,
  |- t \in T  ->  t ===> t'  ->  |- t' \in T.
QuickChick preservation.
```

This simulates a common workflow of Coq users: in order to prove a large theorem (e.g. type safety), one often `Admits` smaller lemmas to construct the proof, discharging them afterwards. However, admitting a lemma that is too strong can lead to a lot of wasted effort and frustration. Using QuickChick, users can uncover bugs early on while building confidence in such conjectures.

For a small portion of the theorems we allowed minor changes (i.e., converting preconditions like `beq_nat n1 n2` to `n1 = n2`). Overall, we only performed one major change: converting Software Foundation `Maps` from a functional representation to an explicit list-based one. A functional representation for maps is convenient for looking up element's associations, but—unless the domain of the function is bounded—makes it completely impossible to do the reverse. That requirement is very common in generation—for instance, picking a variable with a specific type from a given context. Moreover, a lot of properties needed to decide equivalence of two maps, which is also impossible in a functional representation. Therefore, we changed maps to a more generation-friendly variant. The new map code was similar in length with respect to the old one ($\sim$ 40 lines), including automatic derivations of generators and decidability instances, but resulted in many syntactic changes across the rest of the chapters.

5.4.2 QuickChecking Noninterference

To evaluate the efficiency of our approach, we conducted a case study comparing the runtime performance of our derived generators against carefully tuned handwritten ones: the generators for the information-flow control experiments in Chapter 3, which generate indistinguishable pairs of machine states using QuickChick to discover noninterference violations. We reused the mutation testing methodology

to systematically evaluate our derived generators and ensure they had roughly the same bug-finding capabilities. Our experiments showed that the derived generators were 1.75× slower than the corresponding handwritten ones, while producing the same distribution and bugfinding performance.

To summarize Chapter 3, dynamic information-flow control tags data values with security levels, called *labels*, and uses them to prevent flows from *high* (secret) to *low* (public) data. We enhanced a simple stack machine with label information and tested it for *termination-insensitive noninterference*: given two *indistinguishable* machine states, i.e. states that differ only in high data, running them to completion should yield indistinguishable states. This is a prototypical example of a conditional property: if we were to generate pairs of arbitrary machine states and discard those that are not indistinguishable, we would almost never exercise the conclusion! Instead, we generated a single arbitrary machine state first and then *varied* it to produce a new one that was indistinguishable (by construction).

For our evaluation, we focused on a stronger property (also considered in Chapter 3), *single-step noninterference*, which only runs both machines for a single step. As we saw, this makes generators for valid initial states substantially simpler: since only one instruction will be executed, memories do not need to be longer than two elements (no more than one element can be accessed by each machine), integer values that are valid pointers are only 0 or 1 (since the memories are two elements long), and stacks do not need to be large either.

Consider, for instance, a generator for stacks (of a given length `n`), which can be empty (`Mty`), cells that store a tagged integer (`Cons`), or specially marked stack frames that store a program counter to be used by a future `Return` instruction (`RetCons`); `gen_atom` produces mostly in-bounds tagged integers.

```
Fixpoint gen_stack (n : nat) : G Stack :=
  match n with
    | O => returnGen Mty
    | S n' =>
      freq [ (10, liftGen2 Cons gen_atom (gen_stack n'))
           ; (4,  liftGen2 RetCons gen_atom (gen_stack n')) ]
  end.
```

The behavior of this generator can be described by a simple inductive predicate, where `good_atom` describes the behavior of `gen_atom`.

```
Inductive good_stack : nat -> Stack -> Prop :=
| GoodStackMty  : good_stack 0 Mty
| GoodStackCons : forall n a s ,
    good_atom a  -> good_stack n s ->
    good_stack (S n) (a  :: s)
| GoodStackRet  : forall n pc s,
    good_atom pc -> good_stack n s ->
    good_stack (S n) (RetCons pc s).
```

Finally, we can achieve the same distribution with a weight annotation before deriving generators.

```
QuickChickWeights [(GoodStackCons, 10); (GoodStackRet, 4)].
Derive ArbitrarySizedSuchThat for (fun s => good_stack n s).
```

The implicit assumptions for single-state generators are encoded in inductive predicates, and the indistinguishability relation is used to derive variation generators.

We tested the single-step noninterference property 10000 times using both the handwritten and the derived generators. Our derived generators were 1.75× slower than the handwritten ones, while both generators uncovered all mutants successfully. To ensure both generators yield similar distributions of inputs, we used QuickChick's `collect` to determine the number of times each instruction was generated during those 10000 tests (as this was the metric that was used to fine-tune the handwritten generators in the first place).

The observed 1.75× slowdown is mostly due to the added overhead of local backtracking and extraneous matches like the one in the `goodTree` example of Section 5.1. A few local optimizations (like pulling a match outside of a call to `backtrack`) could further improve on our performance, but would require additional work to produce the corresponding proof terms. Still, this overhead is much better than the order-of-magnitude overhead of interpreted approaches like Luck. Finally, what we gain in return for this loss in performance is that the declarative nature of the inductive predicates exposes exactly what assumptions are made

about the generated domain, while the produced proofs guarantee completeness for that domain.

5.4.3 QuickChecking STLC

To conclude our evaluation, we will turn once again to the simply typed lambda calculus. The original paper on proving generators correct in QuickChick [105], included a similar case study. In particular, in that paper we modeled STLC in Coq using a standard representation of lambda terms to include standard constant natural numbers.

```
Inductive term : Type :=
| Const : nat -> term
| Id : var -> term
| App : term -> term -> term
| Abs : term -> term.
```

The types of the calculus are just natural numbers and arrows:

```
Inductive type : Type :=
| N : type
| Arrow : type -> type -> type.
```

To represent environments we used lists of `type`s and variables are natural numbers, indexes into this list. The rest of the typing relation is entirely standard and shown in Figure 5.6.

The original case study included a sized generator for generating simply typed lambda terms satisfying this typing relation and a sequence of buggy definitions of the `step` relation. We then uncovered these bugs by testing two conditional properties, progress and preservation, both of which use this typing relation as a precondition. Finally, we proved in the QuickChick framework the correctness of these generators with respect to the definition above.

We repeated this case study for the *derived* generators presented in this Chapter. In particular, we attempted to derive generators for the typing relation above, as well as all the simple inductive types (`type` and `term`) as necessary. We only needed

```
Inductive typing (e : env) : term -> type -> Prop :=
| TId :
    forall x tau,
      nth_error e x = Some tau ->
      typing e (Id x) tau
| TConst :
    forall n,
      typing e (Const n) N
| TAbs :
    forall t tau1 tau2,
      typing (tau1 :: e) t tau2 ->
      typing e (Abs t) (Arrow tau1 tau2)
| TApp :
    forall t1 t2 tau1 tau2,
      typing e t1 (Arrow tau1 tau2) ->
      typing e t2 tau1 ->
      typing e (App t1 t2) tau2.
```

Figure 5.6: STLC in Coq

to alter a single line, to remove the function call. We changed the precondition in the `TId` rule from:

```
nth_error e x = Some tau
```

to

```
bind e x tau,
```

where `bind` is an inductive relation that represents indexing into the list.

With that single change we were able to easily derive the appropriate generators and uncover the same set of (artificially induced) errors. The code necessary was an order of magnitude less than in the handwritten approach (10 lines vs. 150) and resulted in practically immediate feedback (compared to the 2 days that were devoted in the original case study).

5.5 Conclusion and future work

In this chapter, we presented a narrowing-based algorithm for compiling dependently typed inductive relations into generators for random data structures satisfying these relations, together with correctness proofs. We implemented it in the Coq proof assistant and evaluated its applicability by automatically deriving generators to test the majority of theorems in *Software Foundations*.

In the future, we aim to extend our algorithm to a larger class of inductive definitions, as well as adapt more ideas from Luck. For example, incorporating function symbols is straightforward: simply treat functions as black boxes, instantiating all of their arguments before treating the result as a *fixed* range. For statically known functions, we could also leverage Coq's open term reduction to try to simplify function calls into constructor terms. Finally, it would be possible to adapt the established narrowing approaches for functional programs to meaningfully instantiate unknown function arguments against a known result pattern, just like in Luck.

We also want to see if our algorithm can be adapted to derive decidability instances for specifications in `Prop`, allowing for immediate, fully automatic testing

feedback. We are also interested in *shrinkers* for constrained data, to complete the property-based testing ecosystem for Coq.

Acknowledgments

The work presented in this Chapter was published in POPL 2018 in the paper with the same title "Generating Good Generators for Inductive Relations", with Zoe Paraskevopoulou (who was responsible for the majority of the proof generation framework) and Benjamin Pierce.

Chapter 6

Implementation

In this Chapter we will present the two major side-contributions that emerged out of the implementation aspect of the work presented so far: a generic programming library for Coq and an elegant functional data structure for sampling from updatable discrete probability distributions.

6.1 Generic Programming Framework in Coq

Our initial implementation of the generator derivation algorithm interfaced directly with Coq's internals. But, even for the simply-typed inductive generators we saw in the QuickChick tutorial (Chapter 2), this was neither an extensible nor a maintainable approach. Coq's term data structure, for example, contains far too much type information that our application does not care about. Similarly, the internal functions that produce Coq expressions take more arguments than we need, in order to accurately populate the rich data structure. For example, the (completely illegible) OCaml code that was used to derive a simple `Show` typeclass instance prior to the introduction of our library can be "seen" in Figure 6.1.

To facilitate deriving such instances (generators, proof terms, printers and shrinkers), we wrote a small generic programming framework consisting of two parts: a high level representation of the class of inductive terms we target and a small DSL for producing Coq expressions.

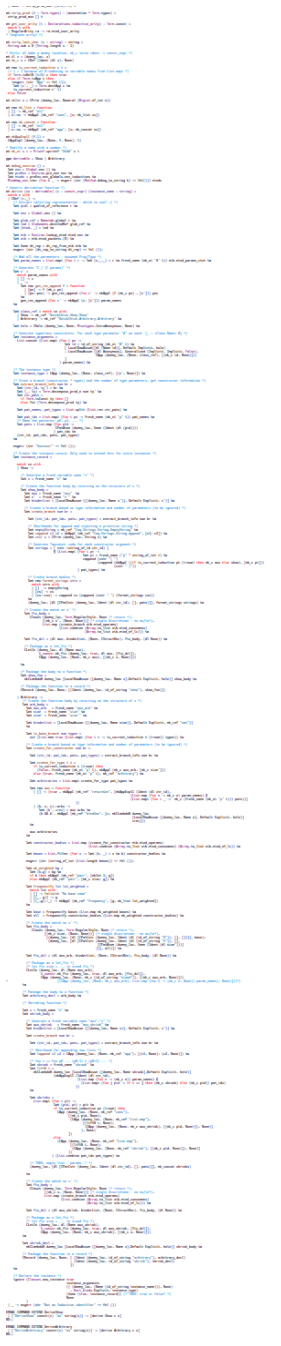


Figure 6.1: Previous OCaml code for deriving QuickChick typeclass instances, prior to the introduction of our library to the QuickChick codebase

6.1.1 Datatype Representation

We represent the class we target with the following datatype:

```
type dep_type =
  | DArrow   of dep_type * dep_type           (* Unnamed arrows   *)
  | DProd    of (var * dep_type) * dep_type   (* Binding arrows   *)
  | DTyParam of ty_param                      (* Type parameters  *)
  | DTyCtr   of ty_ctr * dep_type list        (* Type constructor *)
  | DCtr     of constructor * dep_type list   (* Data constructor *)
  | DTyVar   of var                           (* Type variables   *)
```

The representation is relatively standard. Note that arrows and products are treated as a top-level constructors, since they are of particular importance: arrows can be used to represent side-conditions and products to capture the universally quantified variables of each constructor. Each type above (like `var` or `constructor`) is an opaque wrapper around Coq identifiers, completing the separation of the generic library user from Coq internals.

Using this `dep_type`, we can represent constructors as a pair of an opaque reference to the constructor name and its corresponding type:

```
type ctr_rep = constructor * dep_type
```

Finally, a datatype representation is simply a type constructor, a list of its type parameters and a list of representations of its constructors. Type constructors and type parameters are just opaque wrappers around their corresponding Coq internals:

```
type dt_rep = ty_ctr * ty_param list * ctr_rep list
```

6.1.2 A Term-Building DSL

The second component of the generic library is a DSL for abstracting away from Coq internals when generating terms. For example, the original QuickChick derivation code for a simple `let-fix` declaration inside a `Show` instance contains the following:

```
let aux = fresh_name "aux"  in
let x'  = fresh_name "x'" in
let binderList =
    [LocalRawAssum ([(dummy_loc, Name x')],
                    Default Explicit, c')] in

let fix_dcl = (dl aux, binderList, (None, CStructRec),
               fix_body, (dl None)) in

CLetIn (dummy_loc, dl (Name "aux"),
  G_constr.mk_fix (dummy_loc, true, dl aux, [fix_dcl]),
  CApp (dummy_loc, (None, mk_c aux), [(mk_c x, None)]))
```

Understanding what each term in the expression above does and dealing with the particularities of different binding forms requires diving into the (not really documented) Coq internals. Moreover, even when we got this code to work, maintaining or changing it is virtually impossible. Consider instead the following `gRecIn` combinator:

First, the generic programmer needs to specify the name of the function and its arguments (although they will be made fresh internally to avoid capture). To construct the body of the fixpoint, the programmer should have at their disposal opaque symbols for the function and the arguments and use the various DSL combinators. By using an opaque representation of the various bound terms, we aim to guarantee that every term produce via our combinators is well scoped. Similarly for the body of the let, we should only be able to access the function symbol, not its arguments.

This leads to an important property of our library, and an important design tradeoff we made while constructing it: while we do not guarantee that all generic programs written in our framework produce only well-*typed* terms, we do guarantee that all programs written solely against our interface will produce well-*scoped* terms [1]. We ensure this by making sure that for every combinator that requires the

[1]This presumes that all such generic programs are written in the purely functional fragment of OCaml, as the use of references in particular can defeat this guarantee.

programmer to specify what to do with names, the names are given abstractly. For instance, when the programmer specifies what should occur in the body of the `let rec`, they gives a function of type `(var -> coq_expr)`. Recalling that `var` is an opaque, abstract type, we know that this function must therefore be parametric in its argument. Thus, by strategically using abstract types, we can enforce a kind of poor-person's parametricity, resulting in an interface much like parametric higher-order abstract syntax[28] that can enforce well-scoped-ness, but not well-typed-ness.

Why not make sure that we can only produce well-typed terms from derivers written in our DSL as well? For one thing, OCaml's type system is substantially less powerful than Coq's, so in order to enforce that only well-typed terms are ever produced by OCaml programs written in our library, we would have to encapsulate all of Coq's type system—quite a daunting task.

Beside this technical limitation, we also wanted to hit a sweet spot between ease-of-use for the programmer and enforced correctness. In our opinion, requiring incredibly complex programming with dependent types would not make it quick and easy for programmers to write the kind of derivers we wish them to, at least given the current state of research in dependently typed generic programming. We feel that assisting the programmer in generically writing well-scoped Coq code strikes a happy medium between practicality and provable correctness. In practice, what this means is that if the programmer makes a mistake when implementing a generic deriver using our framework, their code will fail at compile-time in Coq (perhaps only for some particular data types but not others). It's important to note that regardless, type-safety from the perspective of Coq is still preserved—there is no way to use our framework to introduce a bogus type equality to Coq.

Let us return to the library itself. As an example usage, consider the `let-fix` declaration using our combinators:

```
gRecIn "aux" ["x"] (fun aux [x] -> ...) (fun aux   -> ...)
```

In definitions like this, we aim to mirror the syntax of the Coq source being generated as much as possible, so as to make the mental translation burden for the user of the library as light as possible.

Another example of such a combinator is pattern matching. In our current design, `gMatch` has the following type:

```
val gMatch : coq_expr ->
    (constructor * string list * (var list -> coq_expr)) list ->
    coq_expr
```

This combinator takes a Coq expression (the discriminee) and a list of branches, each of which is a constructor (opaque, taken from our datatype representation) a list of strings to name the patterns, as well as a body for each branch with access to the particular pattern variables of that constructors, and produces a Coq term corresponding to the entire expression.

6.1.3 A Worked Example

To illustrate how one might use the library, we now present in its entirety the implementation of the generic deriver for the `Show` typeclass. We will then walk through it step-by-step to explain how it is constructed.

```
let show_body x =

  let branch rec_name (ctr,ty) =
    (ctr, generate_names_from_type "p" ty,
    fun vs -> str_append (gstr (constructor_to_string ctr ^ "  "))
        (fold_ty_vars
            (fun _ v ty' ->
              str_appends [ gstr "( "
                          ; gapp (if iscurrenttyctr ty'
```

```
                                then gvar rec_name
                                else ginject "show")
                            [gvar v]
                        ; gstr " )"
                        ])
            (fun s1 s2 -> str_appends [s1; gstr " "; s2])
            emptystring ty vs))
  in

  gRecFunIn "aux" ["x'"]
    (fun (aux, [x']) ->
      gMatch (gVar x')
        (List.map (branch aux) ctrs))
          (fun aux -> gApp (gVar aux) [gVar x])
in

let show_fun = gFun ["x"] (fun [x] -> show_body x) in
gRecord [("show", show_fun)]
```

Because OCaml does not have `where`-clauses, the nested structure of this definition is presented in a somewhat upside-down manner. As such, we shall examine it from the bottom up.

A typeclass in Coq is merely a record mapping method names (as field names) to their implementations (as values of that field). In the case of the `Show` typeclass, we need to produce a record with exactly one field, named `show`, which is bound to a function of type `(T -> string)` for whatever type `T` we are deriving `Show` for.

```
gRecord [("show", show_fun)]
```

To do this (above), we invoke the `gRecord` function, which takes a description of the contents of a record and returns a Coq term corresponding to the actual record.

We then define the function which is bound to the `show` method.

```
let show_fun = gFun ["x"] (fun [x] -> show_body x) in ...
```

The function takes one argument, `x`, and has a body equal to whatever `show_body` is (given that abstract variable). It's worth noting here that one piece of lightweight dependent typing would be helpful in improving the library interface: it should always be the case that the list of concrete names passed to `gFun` be the same length as the list of abstract names provided to its function argument. We do not statically describe or enforce this, and it would be nice to use length-indexed lists to do this in future.

So what does the body of the `show` function consist of? Well, it's a (potentially) recursive function of one parameter. We define it as below.

```
gRecFunIn "aux" ["x'"]                          (* 1 *)
  (fun (aux, [x']) ->                           (* 2 *)
    gMatch (gVar x')                            (* 3 *)
      (List.map (branch aux) ctrs))             (* 4 *)
  (fun aux -> gApp (gVar aux) [gVar x])         (* 5 *)
```

By lines, we:

1. define a recursive function `aux` of one argument `x'`
2. ...
3. which matches on that argument,
4. and has a case for each constructor of the data type, where the RHS of the case is determined by the `branch` function (discussed below)
5. and then apply that function to the variable x (passed in from above).

There are two present unknowns in the above code: the list `ctrs`, and the function `branch`. The `ctrs` list is provided to us by initialization code not shown in the example which uses our library function `coerce_reference_to_dt_rep`. This function takes a native Coq internal type and gives us back a `coq_type`, as well as a variety of other information about it, including a list of its constructors, here bound as `ctrs`.

As for the `branch` function which defines what to do for each possible constructor of the data type we are `showing`: its logic is relatively simple as well, but let's break it down into pieces to clarify just how it works.

```
let branch rec_name (ctr,ty) =
  (ctr, generate_names_from_type "p" ty,
  fun vs -> str_append (gstr (constructor_to_string ctr ^ "  "))
      (fold_ty_vars
          (fun _ v ty' ->
            str_appends [ gstr "( "
                        ; gapp (if iscurrenttyctr ty'
                                then gvar rec_name
                                else ginject "show")
                           [gvar v]
                        ; gstr " )"
                        ])
          (fun s1 s2 -> str_appends [s1; gstr " "; s2])
          emptystring ty vs))
```

First, for our library, any case of a match is specified by a triple of a constructor, a list of concrete bound pattern variables names, and a function from (opaque) variables to an RHS. The first two elements of this triple are quick to write:

```
(ctr, generate_names_from_type "p" ty, ...)
```

Here, `generate_names_from_type` creates the appropriate number of names, prefixing them with a given string and suffixing them with unique numbers to freshen them.

For each RHS of the match, we need to construct a function from opaque variables to resultant Coq expressions. Recall that `show : T -> string` for some `T`—so ultimately, we need to return a string. Now we are at the meat of the problem: given a value built of a particular constructor of a particular data type, how do we convert it into a string representation generically?

Let us consider the particular case of our running example: `Tree`s. When we want to `show` a `(Node x l r)`, we need to first make a string for the constructor,

then prefix it to the space-interpolated concatenation of the parenthesized showings of each of the `Node`'s arguments. But how to `show` each of these arguments? If, in the case of `x`, such an argument is not another `Tree`, we merely appeal to the `Show` instance of that type. On the other hand, if—as in the case of `l` and `r`—an argument *is* another `Tree`, we need to make a recursive call to *this* `show` function. This distinction is necessitated by the way that Coq's typeclass mechanism functions: the existence of an instance for a given type is not accessible within the definition of that very instance.

All the information we need to generically define such a show-function is present in the type of the data constructor whose case we are defining. If we consider such a type `(A -> B -> C -> T)` as a list of types to the left of an arrow `[A, B, C]`, then we can fold across this list to obtain the desired function. This produces an expression which, for each variable bound by the pattern match, calls the appropriate `show` function (either recursive or non-recursive) and wraps the result in parentheses, concatenating the results by interpolating spaces.

```
(fold_ty_vars
    (fun _ v ty' ->
      str_appends [ gstr "( "
                  ; gapp (if iscurrenttyctr ty'
                          then gvar rec_name
                          else ginject "show")
                      [gvar v]
                  ; gstr " )"
                  ])
    (fun s1 s2 -> str_appends [s1; gstr " "; s2])
    emptystring ty vs))
```

And there you have it: in only a few more lines than the non-generic `Show` instance, we have defined a generic deriver for `Show` instances that works on all polynomial recursive types. We used this generic library to derive all generators and proofs described in the previous Chapter, a task that seems impossible without such a framework.

Acknowledgments

This generic library was first introduced as a semester project for Penn's "Advanced Programming Languages Seminar" (CIS 670) in the fall semester of 2016. The writing is adopted from the final report of that project (written with Kenneth Foner) and the implementation section of the POPL 2018 paper that formed the basis of the previous Chapter ("Generating Good Generators for Inductive Relations" [82]).

6.2 Urns

A common theme throughout this thesis has been the need to sample from and update a discrete probability distribution efficiently. In both QuickCheck and QuickChick, the ability to fine-tune the distribution of generated values is mostly accomplished using `frequency`. In the information-flow-control experiments of

Chapter 3, we used `frequency` to skew the distribution of instructions to obtain longer executions. In Luck 4, implementing weighted random backtracking required both sampling from a discrete distribution and removing the selected option. In QuickChick 5, similar functionality is necessary to implement the `backtrack` combinator.

To begin with, QuickCheck (and QuickChick) implement `frequency` using a list-based representation of the distribution (`[(Int, Gen a)]`). While simplistic, this list-based representation has been used successfully for a long time: while the asymptotics of sampling from a list-based representation seem inefficient, the common inputs to `frequency` are small, and so the linear-time traversal of the list is inconsequential.

Unfortunately, this representation is not powerful enough when working with *updatable* distributions like the ones the `backtrack` combinator represents: sampling without replacement is only interesting if repeated, which leads to repeated traversals. Moreover, each traversal requires modifying the list to update the distribution. This produces quadratic time (and space) overhead, which can lead to noticeable slowdowns even with relatively small distributions.

In this Chapter, we present the *urn*, an immutable persistent data structure that supports efficiently sampling from a distribution, as well as efficiently updating it: inserting new (weighted) values, removing them, or updating their weights. We avoid the usual complexity of traditional self-balancing search trees because we do not need to keep values in a specific order. Instead, we keep the tree maximally balanced at all times using a single machine word of overhead: its size. Urns provide an alternative interface for the `frequency` and `backtrack` combinators, that allow for efficient sampling of dynamically-updated distributions. We empirically evaluate the different versions using examples from the literature.

6.2.1 Sampling Discrete Distributions

Back in your introductory math classes, you may have encountered word problems about urns containing balls of different colors – like the urn in Figure 6.2 – where you had to calculate the probability of ending up with specific colors after a few

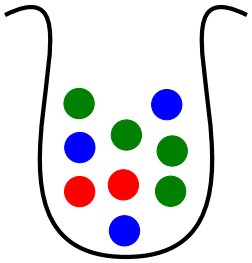


Figure 6.2: A Sample Urn

draws:

> Suppose you have an urn containing two red balls, four green balls, and three blue balls. If you take three balls out of the urn, what is the probability that two of them are green?

This process, often referred to as sampling without replacement, can be seen as a particular instance of a more general problem: sampling from *updatable discrete distributions*.

At their core, urns represent such discrete distributions. We can represent a discrete distribution D over a set A as a nonempty set of pairs of positive weights ${w_i}^+$ and of values $x_i A$; that is,

$$D = \{(w_1, x_1), \ldots, (w_n, x_n)\}({}^+A),\ n1.$$

Let $W = \sum_{i=1}^{n} w_i$ be the sum of all the weights; then, to sample a value from D is to pick a random x_k with probability w_k/W.

To sample from such a distribution, we use the range $[0, W)$ as indices into it. If we pick a natural number uniformly at random from $[0, W)$, we can map it to the x_i: the first w_1 natural numbers correspond to x_1, the next w_2 natural numbers correspond to x_2, and so on. Intuitively, we are breaking the range $[0, W)$ into n buckets: $[0, w_1)$, $[w_1, w_1 + w_2)$, and so on up through $[w_1 + + w_{n-1},\ w_1 + + w_{n-1} +$

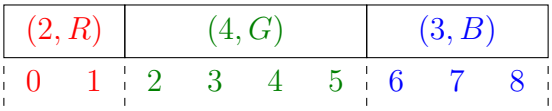


Figure 6.3: Indexing into the discrete distribution $\{(2, R), (4, G), (3, B)\}$; natural number indices are placed below their corresponding weight-value pair (the order is arbitrary).

$w_n = W$). Then the kth bucket, which corresponds to x_k, is

$$\left[\sum_{i=1}^{k-1} w_i,\ \sum_{i=1}^{k} w_i\right)$$

and has size

$$\sum_{i=1}^{k} w_i - \sum_{i=1}^{k-1} w_i = w_k.$$

Thus, there are w_k different values for each index that result in picking this bucket. Since each index is equally likely, the total probability of picking x_k is w_k/W. An instantiation of the buckets for the distribution $\{(2, R), (4, G), (3, B)\}$ – which will be a running example for the remainder of the paper – appears in Figure 6.3.

This approach is the basis of the urn sampling algorithm, as well as the standard `frequency` combinator in QuickCheck and QuickChick, and the array-based binary search variant used in `random-fu` [119] (for more about the latter two, see Section 7.5).

6.2.2 The Urn Data Structure

Since urns represent discrete distributions their API must include support for

(a) *constructing* urns from a list of pairs of weights and values á la `frequency`

(b) *sampling* from urns

(c) *modifying* urns, which there are three ways to do:

  (i) a variant of sampling that removes the sampled value from the urn;

```
data Urn a                          -- a discrete distribution; abstract
type Weight = Word                  -- nonzero
class Monad m => MonadSample m      -- provides randomness
```

```
singleton   ::  Weight -> a -> Urn a
fromList    ::  [(Weight,a)] -> Maybe (Urn a)
```

```
sample      ::  MonadSample m => Urn a -> m a
remove      ::  MonadSample m => Urn a -> m (Weight, a, Maybe (Urn a))
```

```
insert      ::  Weight -> a -> Urn a -> Urn a
update      ::  MonadSample m => (Weight -> a -> (Weight, a)) -> Urn a
            ::                  -> m (Weight, a, Weight, a, Urn a)
replace     ::  MonadSample m => Weight -> a -> Urn a
            ::                  -> m (Weight, a, Urn a)
```

```
size        ::  Urn a -> Word
totalWeight ::  Urn a -> Weight
```

Figure 6.4: The API for urns: the types, constructors, sampling functions, and updating functions.

(ii) the ability to insert new (weighted) values into the urn;

(iii) the ability to update the weights and values found in the urn.

The full API for urns is presented in Figure 6.4; the interface is split into five categories.

Types These include `Urn`, the type of discrete distributions; `Weight`s in those distributions; and `MonadSample`, a type class for monads that support random number generation to enable sampling from urns, such as `IO` or QuickCheck's `Gen` type for random generators. [2]

Construction The `singleton` and `fromList` functions create distributions where the given values (of type `a`) have the corresponding weights. The `fromList` function produces a `Maybe (Urn a)` because distributions cannot be empty.

[2]While the `MonadRandom` type class [47] would seem to be a good fit for this purpose, `Gen` is unfortunately not an instance of it.

Sampling The `sample` and `remove` functions both pick a value from the `Urn` at random, with probability proportional to its weight. The `remove` function also takes that value out of the urn ("sampling without replacement"), and so returns the `Weight` of that value and `Maybe` the updated `Urn` . Note that `remove` takes out the whole weight-value pair, rather than taking out one copy of the value and reducing its weight by 1.

Modification The `insert` function simply adds a new value to the distribution with a given weight. The `update` and `replace` functions both choose a random value to modify, as per `sample` . The `update` function will modify that value and its weight, returning the old weighted value, the new weighted value, and the new `Urn` ; the `replace` function simply overwrites the chosen value and its weight, returning the old weight and value along with the new `Urn` .

Properties `Urn`s keep track of how many values they contain (`size`) and their total weight (`weight`).

Coding Conventions As we saw in Section 6.2.1, sampling from a distribution D with total weight W can be done by sampling a natural number uniformly from $[0, W)$ and using it as an index into D. The `MonadSample` type class provides the index-generation functionality; its only method is

```
randomWord :: MonadSample m => (Word,Word) -> m Word
```

where `randomWord (low,high)` generates a random `Word` chosen uniformly from the range `[low,high]`.

In the remainder of this section, we implement all the functions from Figure 6.4 that require randomness by phrasing them instead in terms of indices into the urn. Every such function now requires an additional argument of type `Index`, where `Index` is a type synonym for `Word`. When formulated this way, these functions are deterministic. Thus, although the user-facing version of `sample` has type `MonadSample m => Urn a -> m a`, the implementation that we show in this section has type `Urn a -> Index -> a`, and similarly for `remove`, `update`, and

`replace`. We connect the randomized and deterministic versions of these functions by generating random indices with `randomWord (0, w-1)`, where `w` is the total weight of the input `Urn`.

6.2.3 A Weighty Matter

The `Urn` abstract data type, behind the scenes, is implemented as a balanced binary tree – the functional programmer's go-to choice for logarithmic-time operations. However, before we consider how the trees are balanced, we need to consider how they represent discrete distributions in the first place; we save balancing concerns for Section 6.2.5.

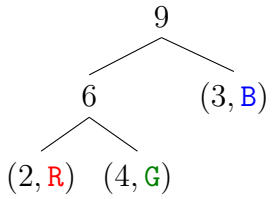
As distributions must be nonempty, we use a nonempty binary tree that stores data – values in the distribution – at the leaves. In addition, we must also store information about the weights of each value: each location in the tree, leaf and (internal) node alike, stores a weight. We maintain the invariant that the weight of a tree or subtree is the total weight of every value in the corresponding distribution. This means that the weight of a leaf is simply the weight of the value it holds, and the weight of a node is the sum of the weights of its children. Such a tree can be represented by the following data type:

```
type Weight = Word

data Tree a = Node Weight (Tree a) (Tree a)
            | Leaf Weight a

weight :: Tree a -> Weight
```

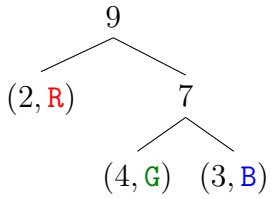
Our example distribution, $\{(2, R), (4, G), (3, B)\}$, can be represented as a `Tree` in multiple different ways by altering the grouping or the ordering of values. Three possible tree representations of this distribution are shown in Figure 6.5.

The rationale behind storing the aggregate weights at the internal nodes comes from thinking about the buckets from Section 6.2.1. We can think of each `Node w l r` as representing a single "super-bucket" of size `w` , where the "super-bucket" spans the buckets of every value at the leaves. If the total range covered by

```
-- Simple tree representation
Node 9 (Node 6 (Leaf 2 R)
               (Leaf 4 G))
       (Leaf 3 B)
```




```
-- An alternate grouping
Node 9 (Leaf 2 R)
       (Node 7 (Leaf 4 G)
               (Leaf 3 B))
```




```
-- Order doesn't matter
Node 9 (Node 5 (Leaf 2 R)
               (Leaf 3 B))
       (Leaf 4 G)
```

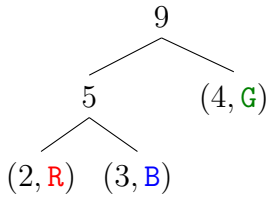


Figure 6.5: Three different tree representations of the distribution $\{(2, \mathtt{R}), (4, \mathtt{G}), (3, \mathtt{B})\}$.

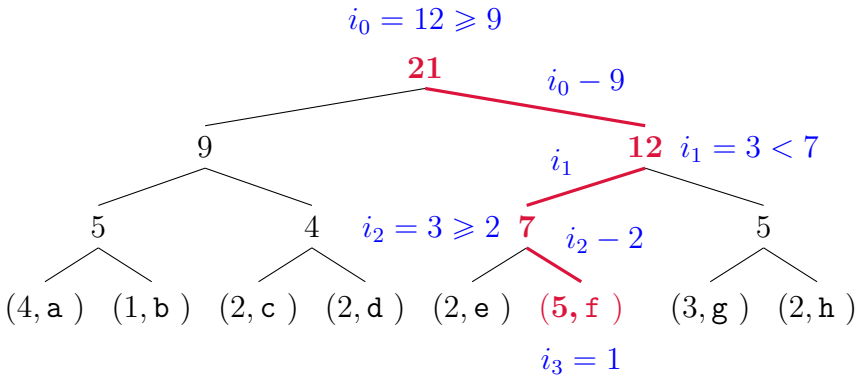


Figure 6.6: What happens when sampling from an urn. This example looks up the index 12 in a tree representing the distribution $\{(4, \texttt{a}), (1, \texttt{b}), (2, \texttt{c}), (2, \texttt{d}), (2, \texttt{e}), (5, \texttt{f}), (3, \texttt{g}), (2, \texttt{h})\}$. The path taken through the `Tree` is in **bold red**; the changes to the index i_x at the xth recursive call are in blue. As this shows, adjusting is only done when recursing into the right-hand child of a node.

this tree is $[b, b + \texttt{w}\ )$, then its two subtrees `l` and `r` split it into $[b, b + \texttt{wl}\ )$ and $[b+\texttt{wl}\ , b+\texttt{wl}\ +\texttt{wr}\ )$, where `wl` = `weight l` , `wr` = `weight r` , and `wl` + `wr` = `w` by the invariant on weights. An index `i` into this range falls in the left super-bucket if `i < b + wl`, and the right super-bucket otherwise; applying this recursively, we end up in the correct bucket, which is to say at the correct leaf. This algorithm can be slightly simplified by always adjusting the super-buckets to start at 0. This allows every `Tree` to be considered in isolation, without any need to keep track of the super-bucket's base b; to do so, we simply adjust `i` if it would fall in the right-hand bucket, subtracting `wl` . This leads to the following Haskell implementation:

```
sample :: Tree a -> Index -> a
sample (Leaf _ a)   _ = a
sample (Node w l r) i
  | i < wl    = sample l i
  | otherwise = sample r (i - wl)
  where wl = weight l
```

The result of running this algorithm on an 8-leaf `Tree` is presented in Figure 6.6.

6.2.4 Turning Over a New Leaf

The `update` and `replace` functions from Figure 6.4 are similar to `sample` , but they return a modified `Urn` in addition to the randomly chosen value. We consider `update` first: given a function `upd :: Weight -> a -> (Weight,a)` and an urn `u` , the call `update upd u` randomly chooses a value `a` in the urn with some weight `w` , applies `upd w a` to get the result `(w',a')` , and returns a triple `((w,a), (w',a'), wt')` consisting of the old weighted value, the new weighted value, and the new `Urn` , which has had `w` and `a` replaced by `w'` and `a'` . (We return both `(w,a)` and `(w',a')` in case we need the new values, as this avoids recomputing them when `upd` is expensive.)

The way that `update` uses an index to traverse a tree is the same as `sample`. However, as `update` modifies the `Tree` , we need to update the weights as we rebuild the tree on the way back up: every weight above the updated leaf must be adjusted by the difference `w'-w` . We do not need to worry about rebalancing, since the *structure* of the `Tree` and the number of values it contains is unchanged.

```
update :: (Weight -> a -> (Weight,a)) -> Tree a
       -> Index -> ((Weight,a), (Weight,a), Tree a)
update upd (Leaf w a)   i =
  let (w',a') = upd w a
  in ( (w,a), (w',a')
     , Leaf w' a' )
update upd (Node w l r) i =
  | i < wl =
    let (old, new, l') = update upd l i
    in ( old, new
       , Node (w - fst old + fst new) l' r )
  | otherwise =
    let (old, new, r') = update upd r (i-wl)
    in ( old, new
       , Node (w - fst old + fst new) l r' )
  where wl = weight l
```

We elide the implementation of `replace w' a'`, as it is essentially the same as `update (_ _ -> (w',a'))`; the difference is that since `w'` and `a'` are statically known, we only need to return a pair `((w,a), wt')` containing the old weighted value and the new urn.

6.2.5 A Balancing Act

As mentioned at the beginning of Section 6.2.3, if we want logarithmic runtime for all our operations, we need to make sure our trees stay balanced when we add or remove values from the distribution. However, the `Tree` type presented thus far does not contain enough information to stay balanced if we change the size or layout dynamically. Because there is no natural ordering to the values contained within an urn, using a self-balancing binary search tree such as an AVL or red-black tree is unnecessarily complex and a poor fit for the problem we wish to solve. Such an implementation would force us to impose an ordering on the values contained in the urn, and some values we frequently wish to store in an urn – such as QuickCheck generators, which are wrappers around functions – cannot be given an ordering at all.

The key insight to balancing `Tree` s in the simplest way is to realize that, unlike for binary search trees, *order is truly irrelevant*; we first encountered this in Figure 6.5. The efficiency of `sample` also does not depend on ordering, as the only invariant we have imposed on our `Tree`s is that `weight (Node w l r) == weight l + weight r` . Thus, if we always insert values at the second-deepest level of the tree until we must start a new level, we will maintain the balance.

When we wish to insert a new value into the tree, we take some path to get there, which involves going left or right at each `Node`. If we always go in the *opposite direction* on each successive insertion, we will distribute our updates evenly throughout the tree. We can do this by storing a *direction* at each `Node` : either left (←) or right (→). To decide where to insert, we recurse into the child we are directed to, and toggle the direction.

This allows us to implement the self-balancing insertion function `insert :: Weight -> a -> Tree a -> Tree a`, , where `insert w a` inserts the value `a` into

a distribution with weight `w` . As we go down the tree, we add the to-be-inserted weight `w` to the weight at every node we pass; to decide which way to go, all we need to do is follow the directions.

It is easiest to see what this means by looking at how this approach iteratively builds up a new tree from a singleton distribution, one insertion at a time; we present an example of this in Figure 6.7. Each successive tree in the figure has a new leaf at the location found by following the arrows down from the root in the previous tree (on the right of the old leaf), and all the arrows that were followed in that traversal are flipped from said previous tree. We can see that this "evenly distributes" new leaves, rather than filling them in from left to right.

6.2.6 Losing Direction

We can look at the insertion pattern shown in Figure 6.7 and record the directions we take, using `L` for left and `R` for right; every such path ends with an `R`, as new leaves are added to the right of old ones. The sequence of insertions we get is shown in the following table:

The pattern of `L`s and `R`s is a familiar one: if `L` is 0 and `R` is 1, then we can read any given path backwards as a binary number. Enumerating our paths in this way counts from $1_{10} = 1_2$ through $7_{10} = 111_2$. This means that the path we must take to find a new insertion location is given exactly by the binary representation of the *size* of the `Tree` before insertion!

Thus, all we need beyond our original `Tree` type is a single `Word` keeping track of the size, and we have all the balancing information we need. It is this composite data type consisting of a size and a `Tree` that we call an `Urn`:

```
data Urn a = Urn { size :: Word
                 , tree :: Tree a }
```

All the functions that we defined on `Tree` s are lifted to `Urn` s by operating on the `tree` field.

This also saves space! A `Tree` holds the minimum amount of information that we need to sample from a discrete distribution; if we had to include directions in a `Tree` with n values, we would incur $O(n)$ overhead to store them. With an `Urn` ,

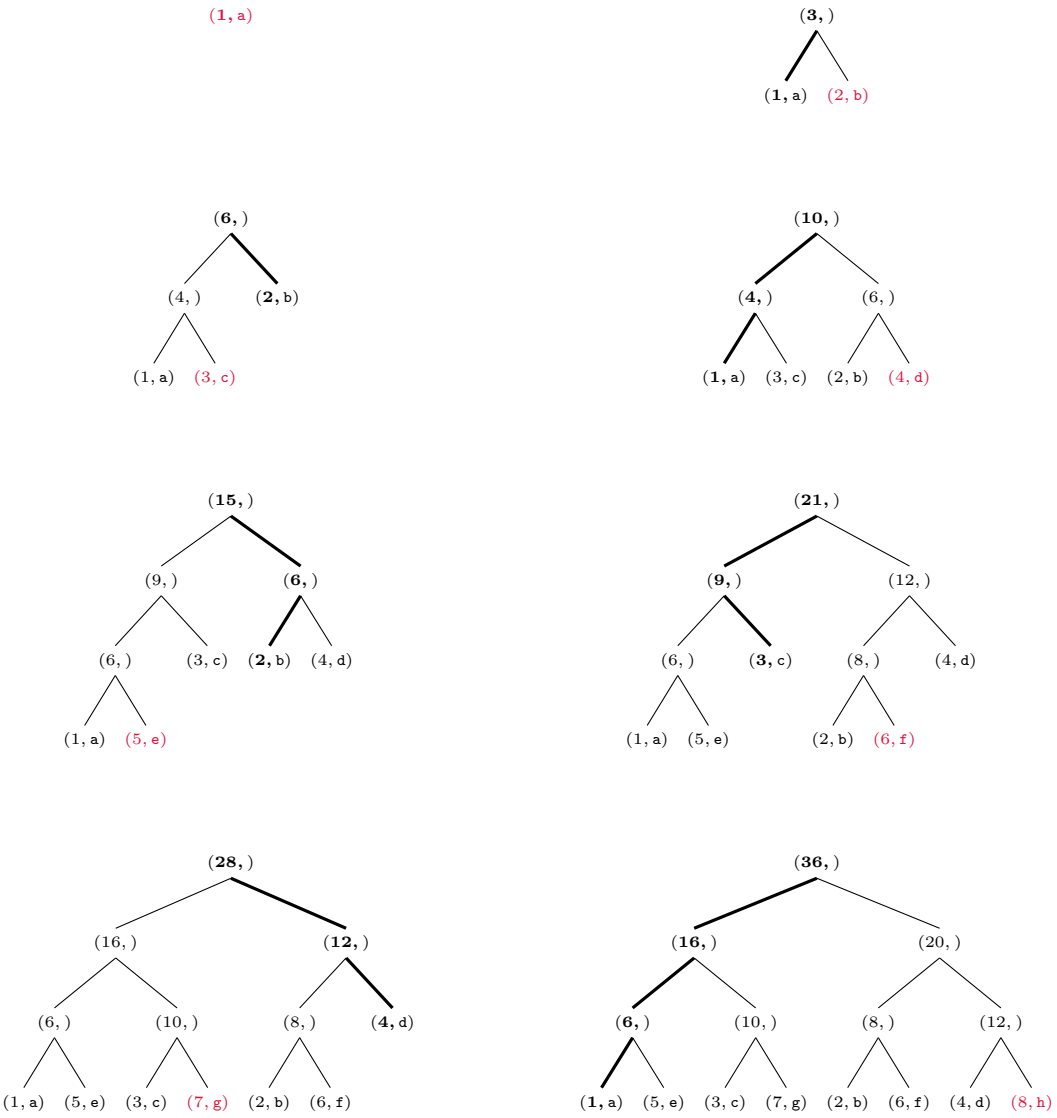


Figure 6.7: Iteratively constructing a directed tree, building up the distribution $\{(1, \texttt{a}), (2, \texttt{b}), (3, \texttt{c}), (4, \texttt{d}), (5, \texttt{e}), (6, \texttt{f}), (7, \texttt{g}), (8, \texttt{h})\}$ one value at a time. The **bold** paths indicate where the next value will be inserted; the red leaves are the most-recently-inserted leaf in each tree.

our space overhead, with respect to an ordinary tree representation, is reduced to a single machine word.

We have to change the insertion algorithm to use the size of the urn to perform traversal instead of embedded directions. At every step, the low bit of the given size is the direction to travel – if the bit is `1` , we go right, and if it is `0` , we go left. In the recursive call, we shift off the lowest bit and recurse, getting access to the next lowest bit, which is the next direction in the path.[3]

```
insert :: Weight -> a -> Urn a -> Urn a
insert w' a' (Urn size tree) =
  Urn (size+1) (go size tree)
  where go _    (Leaf w a) =
          Node (w+w') (Leaf w a) (Leaf w' a')
        go path (WNode w l r)
          | path `testBit` 0 =
            WNode (w+w') l (go path' r)
          | otherwise =
            WNode (w+w') (go path' l) r
          where path' = path `shiftR` 1
```

As we see in `insert`,binary numbers correspond to root-to-leaf paths in an urn read backwards. In fact, for an urn of size n, all binary numbers less than n correspond to valid paths; vice versa, all paths to leaves correspond to binary numbers less than n. Moreover, just like `insert` uses n as the path to the insertion point, we will always be able to use this same n as the path to that location.

6.2.7 A Value Un-urned

Now that we have the definition of `Urn`s, the final piece of functionality we need to implement is deletion. In order to maintain balance in our trees, we cannot remove values from arbitrary locations. There is precisely one node whose removal would leave the tree compatible with further iterated insertion: the most recently inserted

[3]Computing `n `testBit` b` determines whether the `b` th bit of `n` is set.

value. Removing this value would take us back to the previous, also-balanced tree.[4] We call this operation `uninsert` , and we can combine it with `replace` to implement `remove`: we `uninsert` the most-recently-inserted weighted value, and then `replace` the weighted value we want to `remove` with said `uninsert`ed weighted value.

However, as they say, the devil is in the details. The first thing we need to do is call `uninsert` to produce the value we need to pass to `replace` , as well as a new urn. Since urns cannot be empty, `uninsert` actually returns a `Maybe (Urn a)` – `uninsert` ing from an `Urn` of size one produces `Nothing`. Moreover, if the result *is* `Nothing` , we are done: there was only one possible value we could have removed, so we must have removed it.

On the other hand, if the result is a `Just` , we encounter a problem: `remove` had as an argument an index `i` into the urn that we had *before* calling `uninsert` , which means it cannot be used to index into its result – some of the indices may have shifted during uninsertion. We can see a visualization of what happens to the indices after an uninsertion in Figure 6.8. So how can we update `i` to point to the correct place in the new urn? Again, looking at Figure 6.8, we can see that the indices that fall *after* the removed bucket must be shifted down to fill in the uninserted bucket. We also have to address the case where we were supposed to remove the uninserted value; if `i` lay within the removed bucket, then we don't in fact need to call `replace` at all. This accounts for the extra `w'` indices that are valid for the old urn but not the new one.

Thus, `uninsert` must not only return the weight and the value that was deleted, but also enough information to completely identify the removed bucket: its lower bound. The type of `uninsert` is therefore uninsert :: Urn a → ((Weight,a), Weight, Maybe (Urn a))

The implementation of `uninsert` is very similar to `insert`. When inserting a value, we follow the path given by the bits of `size` itself and insert a `Leaf` , updating all the parent weights in the process; this produces an `Urn` of size `size+1`. In contrast, to uninsert a value, we just need to follow the path given by the bits

[4]Even if `update` s have happened in the meantime, recall that changes to the weights do not affect the balance of the tree; only its leaf-node structure affects the balance.

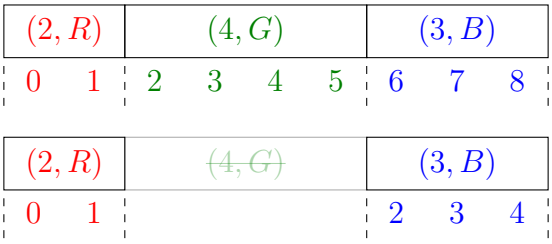


Figure 6.8: What happens to indices when uninserting a bucket: if `uninsert` returns `((4,G), 2, Just ...)`, then the indices into the subsequent B bucket must be shifted down.

of `size-1` and remove the `Leaf` there, again updating internal node weights to maintain the weight-sum invariant.

The bigger difference is that we also need to calculate the lower bound of the bucket of the value we removed. If our tree is just a leaf, `Leaf w a` , then the bucket for that leaf is just `[0,w)` . If our tree is a node, `Node w l r` , then the "super-bucket" for the whole tree is again `[0,w)` , and the two subtrees have "super-buckets" `[0, weight l)` and `[weight l, w)` (as we saw in Section 6.2.3). Therefore, since we know which direction to recurse to find the target, we know how to adjust the lower bound returned by the recursive call. If we recursed to the *left*, then the lower bound is unchanged; if we recursed to the *right*, then we need to add `weight l` to the lower bound.

As promised, we can now combine `uninsert` with `replace` to produce `remove` . This breaks down into the following three cases:

- If `i < lb` , then `i` lies before the removed bucket, so it pointed to the same value in `urn` as it now points to in `urn'` ; thus, we can use the index `i` as-is when calling `replace` .
- If `lb <= i < lb + w'`, then `i` was in the removed bucket, so the uninserted item *was* the very item we had wanted to remove; this means we can return the pair `(old, Just u')` directly without calling `replace` .
- Finally, if `lb + w' <= i` , then `i` lies after the removed bucket, so the value `i` points to has been relocated by `uninsert` ; we need to subtract off `w'` to

get the new index `i-w'` .

The Haskell implementation reflects all of these cases directly.

```
remove :: Urn a -> Index -> ((Weight,a), Maybe (Urn a))
remove urn i =
  let ((w',a'), lb, maybeUrn') = uninsert urn
  in case maybeUrn' of
       Nothing -> ((w',a'), Nothing)
       Just urn'
         | i < lb ->
           Just <$> replace w' a' urn' i
         | i < lb + w' ->
           ((w',a'), Just urn')
         | otherwise ->
           Just <$> replace w' a' urn' (i - w')
```

6.2.8 Building Up To (Almost) Perfection

There is only one more nontrivial function from our API that we have not yet discussed: `fromList`.[5] Having already defined `insert` , we could define `fromList` in terms of it:

```
fromList :: [(Weight, a)] -> Maybe (Urn  a)
fromList ((w,a):was) =
  Just $ foldr (uncurry insert) (singleton w a) was
fromList [] = Nothing
```

Most of the time, this implementation will be fine; it runs in linearithmic – $O(n \log n)$ – time, but since each urn will only be initialized once, this overhead is not a problem in practice. Still, we can do better.

The `fromList` function constructs an urn all at once. Any binary tree with n leaves, such as an `Urn` with n values, also has exactly $n - 1$ internal nodes. This

[5] We elide the implementation of `singleton`.

means that if we could build an `Urn` while spending only constant effort at each node and leaf, we would have a construction algorithm which runs in linear time with respect to the length of the list.

The fold-based algorithm above constructs an urn by iteratively rebuilding it, traversing the tree from root to leaf and modifying its weights and values. Because any such traversal must cost at least logarithmic time, any top-down algorithm must take at least linearithmic time. Instead, to construct an urn in linear time, we need to build it from the bottom-up, starting from the leaves.

A first useful observation is that all urns of a given size have an identical shape – they will only differ in their weights and leaf values. As a result, our construction algorithm need only compute the correct shape for urns with size equal to the length of the input list, summing weights to fill the internal nodes as it goes.

What, then, is the shape of an urn of a given size? We've seen, in Figure 6.7, how this shape evolves as successive elements are inserted. At all times, we maintain the invariant that an urn is *almost perfect* – that is, that the difference in depth between any two leaves is at most one.[6] This means that the shape of an urn is restricted to being composed of a perfect tree with an additional "fringe" of pairs of leaves dangling beneath the last fully perfect row of that tree.

Computing an urn's shape boils down to computing the depth of the deepest full level of the tree, and the positions of all the dangling pairs of leaves beneath that level. Were there no such leaves – that is, were the list size a power of two – then we could build the tree in linear time by simple recursion. We present the algorithm parameterized over an arbitrary tree type `t` with `node` and `leaf` construction functions.

This computation is `State`ful, storing a list of values that will become leaves; the `consume` operation removes and returns the first `n` elements of that list. The structure of the recursion in `perfect` looks like the desired tree, but we still only consume elements from the list one at a time. We recurse on the depth of the desired perfect tree; when this hits zero, we `consume` a single element from the input list and convert it into a leaf. At non-zero depths, we simply recurse twice

[6] This is weaker than the definition of a *complete* tree, which requires in addition that all leaves on the deepest level are as far left as possible.

and produce a node whose children are the two resulting perfect trees.

```
perfect :: (t -> t -> t) -> (a -> t)
        -> [a] -> t
perfect node leaf values =
  evalState values $ go perfectDepth
  where
    size         = length values
    perfectDepth = floorLog2 size

    go 0 = do
      [a] <- consume 1
      pure $ leaf a

    go depth =
      node <$> go (depth - 1)
           <*> go (depth - 1)

    consume :: Word -> State [a] [a]
    consume n = state $ splitAt n
```

Urns, however, are merely *almost* perfect. When the input list is of size $2^d + r$, where $0 < r < 2^d$, we can handle the extra r elements by sometimes consuming two elements at once and building a node with two leaves as children instead of consuming one element and building a leaf. The tricky part is figuring out when to consume two elements. For example, if we wanted to build a *complete* tree, we could consume two elements the first r times, and then one element the remaining $2^d - r$ times. However, the urns built up by `fromList` will not be complete; they must have the shape that would have been produced by repeated insertion.

What we need to determine, then, is whether we consume 1 or 2 values with the ith `consume` action. Recall the invariant from the end of Section 6.2.6: every root-to-leaf path in this urn, read as a string of bits, corresponds to the reverse of a binary number less than 11; at the same time, every binary number less than 11

corresponds to a root-to-leaf path. We only perform a `consume 2` action when the leaves it produces satisfy this invariant. Let's focus on the node containing `d` and `e` , which is produced by the third `consume` action (with index $i = 2$). These two leaves have paths which correspond to $\texttt{0010}_2 = 2_{10}$ (for `d`) and $\texttt{1010}_{\ 2} = 10_{10}$ (for `e`). This was allowed to be a `consume 2` because both 2 and 10 are indeed less than 11. An urn of size 10, on the other hand, would instead contain a leaf (not a node) at this point.

This algorithm is reflected in the `almostPerfect` function below.

```
almostPerfect :: (t -> t -> t) -> (a -> t)
              -> [a] -> t
almostPerfect node leaf values =
  evalState (0,values) $ go perfectDepth
  where
    size         = length values
    perfectDepth = floorLog2 size
    remainder    = size - 2^perfectDepth

    go 0 = do
      ix <- index -- 0 <= ix < 2^perfectDepth
      if reverseBits perfectDepth ix < remainder
        then do [l,r] <- consume 2
                pure $ leaf l `node` leaf r
        else do [a]   <- consume 1
                pure $ leaf a

    go depth =
      node <$> go (depth - 1)
           <*> go (depth - 1)

    index   :: State (Word, [a]) Word
    consume :: Word -> State (Word, [a]) [a]
```

The local variable `size` is the length of the input list, and is equal to $2^{\texttt{perfectDepth}} +$ `remainder`. Calling `reverseBits count n` reverses the lowest `count` bits of the word `n`. Finally, we augment our state with a counter which is incremented every time `consume` is called, and use the `index` action to read its value.

In the code above, we decide whether to `consume 2` or `1` by performing the check `reverseBits perfectDepth ix < remainder`, where `ix` is the index of the current `consume` action. This index is a path to the leaf *or node* that this action will produce. The check we described before would correspond to checking that both the reversals of `ix 0` and `ix 1` are both less than `size`. The first check is always trivially true, as `ix` is less than $2^{\texttt{perfectDepth}}$ (which is also why our invariant is automatically satisfied in the `consume 1` case). For the `ix 1` case, the trailing `1` takes on a value of $2^{\texttt{perfectDepth}}$. We can thus compare the reversal of `ix` without that to `remainder`, which is the number of values beyond $2^{\texttt{perfectDepth}}$

This function does indeed run in linear time: we access each list element once, and we perform a constant amount of work to create each leaf and node.

6.2.9 Applications and Evaluation

Now that we have defined urns, we explore their applications to random testing and benchmark their performance against existing solutions from the literature.

An Alternative `frequency` Combinator

As we've seen in this thesis, the `frequency` combinator allows the user to combine different generators of the same type by choosing one of them based on a discrete distribution. Its implementation is presented in Figure 6.9. Every time `frequency` is called, it calculates the sum `tot` of the weight components of the input list, generates a random number between `1` and `tot` , and indexes into the list linearly. For many applications, the input distribution has only a few values, so this approach is reasonably fast. However, the linear traversal of the list can cause unnecessary overheads for medium-to-large inputs.

We propose a new combinator `frequency'` that takes an `Urn (Gen a)` as input (where `Gen` is QuickCheck's type for random generators), using the random `sample`

function from Figure 6.4.

```
frequency' :: MonadSample m => Urn (m a) -> m a
frequency' = join . sample
```

Its functionality is identical to QuickCheck's `frequency` : we generate a number between zero and the total weight of the urn (`Urn` s are `0` -indexed where `frequency` is `1` -indexed) and index into our structure appropriately. The use of urns provides a lot of flexibility, allowing us to both use the very expressive combinator library of QuickCheck and dynamically change the distributions involved efficiently, as we will see in the rest of this section. Moreover, even in the static case – i.e.,, the case where we do not modify the distribution – we obtain better performance.

In the information-flow control case study [65] we saw in Chapter 3, we explored different generation methods for information flow control stack machines, focusing for the most part on generating "good" instruction sequences; that is, sequences that lead to longer executions. The instructions for their simple machine are: `Push` and `Pop` , which manipulate the stack; `Add`, which sums the top two items on the stack; `Load` and `Store` , which are memory operations; `Jump` , `Call` and `Return` , which are control flow operations; `Halt` , which signifies a successful termination; and `Noop` , which does nothing. The `frequency` combinator is featured prominently in that development: for every generation strategy they explore other than the very first, nave, one, individual instructions are generated using `frequency` .

Benchmarking We evaluated the performance of `Urn`s in the static case by testing one of the early IFC generation strategies (called *genWeighted*), which crucially uses `frequency` to increase the probability of `Halt` and `Push` instructions, skewing the distribution of programs towards those that terminate (`Halt`) and do not crash (because they have a big enough stack to avoid underflows). We randomly generated 500 instructions in the form of instruction lists of size 10, and benchmarked the generation time using Criterion [98]. In this benchmark, sampling from `Urn`s is 2.64× faster than using `frequency` .

To further evaluate `Urn`s, we wanted to identify the cutoff point (in terms of input list size) where using an `Urn` for static sampling becomes more efficient. We

```
frequency :: [(Int, Gen a)] -> Gen a
frequency [] = error "... empty list"
frequency xs0 = choose (1, tot) >>= (`pick` xs0)
 where
  tot = sum (map fst xs0)

  pick n ((k,x):xs)
    | n <= k    = x
    | otherwise = pick (n-k) xs
  pick _ _  = error "... empty list"
```

Figure 6.9: The exact implementation of `frequency` from QuickCheck 2.8.2 (with abbreviated string literals) [33, 108].

used Criterion again to benchmark sampling uniformly from the first n integers (where n ranged from 1 through 10000). For each n, we generated numbers using `frequency` and using `Urn` s; we ran each approach 10000 times with QuickCheck's `sample'` (which generates 11 values using `IO`), and measured the performance. The results appear in Figure 6.10. There was no cutoff: for small distributions ($n \leqslant 20$), the performance of `Urn` s and lists are the same within the margin of error; and for larger distributions, `Urn` s quickly outpace lists. The run time of `frequency` , as expected, scales linearly with the size of the input list, requiring more than 3 seconds to complete when $n = 10000$; on the other hand, the time taken to sample from an `Urn` grows at a much slower rate, rising logarithmically from roughly 50 ms for small inputs to roughly 80 ms for the larger ones. This logarithmic curve can be better seen on the right-hand side of Figure 6.10, where we only plot the time needed to sample from urns.[7]

[7] All the benchmarks in this paper were run on a Dell XPS15 laptop with a 2.3 GHz Intel Core i7-4712HQ with 16 GB of RAM running Ubuntu 16.04.2 LTS; they were compiled with GHC 8.0.2 using `-O2 -funbox-strict-fields`.

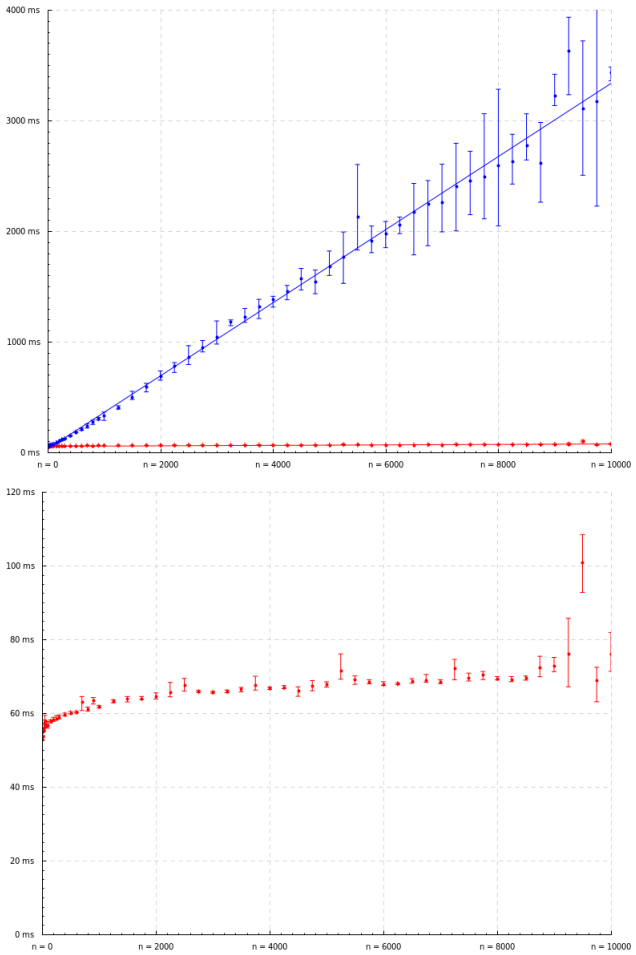


Figure 6.10: Left: Performance of `frequency` (blue, above) vs. `Urns` (red, below). Right: Zoomed-in performance of `Urns`.

An Efficient `backtrack` Combinator

The real benefit of using urns, however, is not just a slight performance boost in the static case. When we wish to dynamically alter the input distribution, urns greatly improve the performance and conciseness of our code. This desire to update a distribution that is being sampled from often arises in random testing when some generators may fail to produce a value (i.e.,, return `Nothing`). If we sample from a generator and the generated value is not a `Just`, we must backtrack and try again.

As an example, consider the inspiring work of Pałka *et al.* [103] on generation of well-typed lambda terms which we've already encounter in the evaluation of both Chapters 4 and 5. To generate well-typed terms, the authors use the typing rules of simply typed lambda calculus as generators. They assign an empirically-chosen weight to each rule; then, to generate a term with type `T` , they pick a rule whose conclusion has type `T` at random based on the weights. This rule may then have premises, which they attempt to satisfy recursively in the same way. For example, to generate a term of type `Int`, we could either use some `Int` constant like `0` or `1` ; some variable `x` from our environment; or a function application `f e'` where `f :: T' -> T` and `e' :: T'` for any type `T'` .

However, the premises of these typing judgments may not be satisfiable. For instance, there might not be any `Int` variables in the environment. Worse, when using the application rules, `T'` is chosen arbitrarily, but there is no guarantee that we can generate a term of type `T'` within the constraints of the generation process. When a typing judgement is not satisfiable, Pałka *et al.* resort to backtracking: they randomly select the next applicable rule. When the remaining rules are exhausted, generation fails and they backtrack at some higher level if possible.

The way Pałka *et al.* choose a rule randomly from a weighted distribution is by permuting the entire list using a variant of permutation-by-sorting shown in Figure 6.11, and then iterating through this permuted list as necessary. Standard permutation-by-sorting shuffles a list by generating a random number for each list item, and sorting the list by comparing these numbers.[8]

[8]One downside of permutation-by-sorting is that it only guarantees a fair shuffle if the generated comparison keys are unique. This is typically avoided by using a fair shuffling algorithm like Fisher-Yates [38]; however, this algorithm does not have a natural extension that takes weights

```
-- Weights must be positive!
permuteWeighted :: [(Int, a)] -> Gen [a]
permuteWeighted xs = do
  v <- mapM (\n  -> replicateM n arbitrary >>=
             \ns -> return $ minimum ns)
            (map fst xs) :: Gen [Int]
  let p = map snd $ sortBy (comparing fst)
                  $ zip v [0..]
  return $ map ((map snd xs)!!) p
  where l = length xs
```

Figure 6.11: The implementation of `permuteWeighted` from Pałka et al. [103] (reformatted).

Pałka *et al.* extend permutation-by-sorting to take (positive, integral) weights w into account by generating w numbers for each item in the list and picking the minimum as the key for sorting. Intuitively, this approach simulates exploding a w-weighted item into w identical copies, using permutation by sorting to shuffle the exploded list and then keeping the first occurrence of each item in the result.

The algorithm in Figure 6.11 has several inefficiencies; apart from implementation details (the use of `!!` could be replaced by `zip` ping with `xs` directly), there are two more fundamental problems. First and foremost, the complexity of the algorithm is pseudo-polynomial; given the weights $\overline{w_i}$, the algorithm runs in $O(n \log n + \sum_i w_i)$ time, since we generate w_i keys for every item before sorting. Secondly, we only need the later items from the shuffled list if we actually backtrack. Thanks to laziness, we may be able to avoid spending the full $O(n \log n)$ time to permute the list, but this depends on the precise sorting algorithm used.

We can avoid completely shuffling a list of generators by putting them in an `Urn`, using the `backtrack` combinator:

into account. Thankfully, the unfairness is not a major concern, since the weights in random testing are typically tuned based on the observed behavior.

```
backtrack :: Urn (Gen (Maybe a)) -> Gen (Maybe a)
backtrack urn = do
  ((_w,g), mUrn') <- remove urn
  ma <- g
  case ma of
    Just a  -> pure $ Just a
    Nothing -> maybe (pure Nothing) backtrack mUrn'
```

In `backtrack`, we `remove` a generator from the urn, sample from it, and test to see if we got a result. If so, we `Just` return that result; otherwise, we repeat this process until the urn is empty. Since each `remove` operation only takes $O(\log n)$ time, this combinator runs in $O(n \log n)$ time (with respect to the running time of the generators).

A similar construct exists in QuickChick [105], but is based on lists; `backtrack` could be used to make this more efficient. In general, analogous situations arise in many other testing applications, such as in explicitly weighted narrowing approaches (e.g., Luck [81]). At a more abstract level, urns can be used to efficiently tune randomized search algorithms that choose between prioritized possibilities.

Benchmarking To evaluate urns in this context, we replaced `permuteWeighted` in Pałka *et al.*'s code [101] with a variation using urns, which inlines the aforementioned `backtrack` combinator:

```
permuteWeighted :: [(Int, a)] -> Gen [a]
permuteWeighted x =
    let Just u = Urn.fromList
               $ map (first fromIntegral) x
        aux u = do
          (w, a, mu) <- Urn.remove u
          case mu of
            Just u' -> (a:) <$> aux u'
            _       -> pure [a]
    in aux u
```

We benchmarked the generation time of 11 Haskell terms (as many as produced by QuickCheck's `sample'`), without altering anything else in their code. Criterion reported a 31.52 ms expected time (1.8 ms variance) for the original code; on the other hand, the urn-based version took 14.95 ms (1.6 ms variance).

While this 2.1× speedup is a victory in its own right, it doesn't measure the real difference between the two variants of `permuteWeighted`. For one, the permutation algorithm is clearly not the bottleneck amongst 1600 lines of complicated Haskell dealing with polymorphic unification. More importantly, because the Pałka variant of `permuteWeighted` has quasi-polynomial running time, it is not general purpose: it is only efficient if all the weights are small. On the other hand, the urn-based variant can be used as-is in any development. Indeed, if we were to directly benchmark the two variants, we could artificially inflate the difference as much as we wanted by choosing arbitrarily large weights.

Acknowledgments

The Urn data structure was published as a functional pearl in Haskell symposium 2017 under the title "Ode on a Random Urn", with Antal Spector-Zabusky and Kenneth Foner, where Antal was primarily responsible for the implementation and its optimization and Kenny came up with the efficient construction algorithm.

Chapter 7

Related Work

7.1 QuickChecks in Theorem Proving

After property-based random testing was popularized by Haskell's QuickCheck [33], it was ported to most programming languages (e.g. [2, 87, 67, 100, 104]). In the particular case of interactive theorem proving, automatically generating counterexamples for false conjectures can prevent wasting time and effort on proof attempts doomed to fail [54]. There have therefore been plenty of QuickCheck-like tools for theorem provers and proof assistants [27, 99, 39, 17, 41, 16].

In this thesis we introduced QuickChick, a property-based testing tool for Coq. The original iteration of QuickChick did not offer much more than the functionality of the original QuickCheck in a new setting; other than dealing with extraction and the totality requirement, QuickChick was essentially a complete clone. We have since improved it with the additional functionality described in this thesis: a framework for proving that generators are correct, and a derivation procedure that produces *correct* generators for data satisfying complex predicates with a customizable probability distribution.

Arguably, the related work closest to QuickChick is the one in Isabelle/HOL. Originally, Berghofer *et al.* [12] proposed a QuickCheck-like tool for Isabelle, which was recently extended by Bulwahn [17]. Bulwahn included support for exhaustive testing as well as a narrowing-based approach. The latter uses Horn clause data

flow analysis to automatically devise generators that only produce data satisfying the precondition of a tested conjecture [18]. While both QuickChick and Isabelle's QuickCheck aim to facilitate verification efforts, there is a fundamental difference in the approach. Just like with most of the tools for finding bugs in Isabelle, its QuickCheck aims to offer a push-button automation experience. This comes with many benefits: it gives immediate feedback to users and allows testing to be run in the background, even while the user is attempting their proofs. However, as we have already discussed earlier in the thesis, automation can only get you so far. There comes a point where a user must leverage domain-specific knowledge to obtain efficient testing by writing custom generators. QuickChick offers the full range of functionality of QuickCheck, including its comprehensive generator combinator library 2. With the generator derivation infrastructure presented in Chapter 5, it edges closer to the automated experience that Isabelle offers to the casual user, while retaining the customizability of QuickCheck, allowing expert users to seamlessly compose derived and handwritten generators together.

Redex [77, 75, 76, 43] (*né* PLT Redex) is a domain-specific language for defining operational semantics within Racket (*né* PLT Scheme), which includes a property-based random testing framework inspired by QuickCheck. This framework uses a formalized language definition to automatically generate simple test-cases. To generate better test cases, however, Klein *et al.* find that the generation strategy needs to be tuned for the particular language. Recently, Fetscher *et al.* [43] combined a narrowing-based algorithm with a constraint solver for equality and disequality constraints to improve upon the automatic generation, achieving excellent performance in testing GHC for bugs (the same case study that was used to evaluate Luck). We discuss the specifics of both this algorithm and the one in Bulwahn's work [18], when addressing the related work of Luck.

Dybjer *et al.* [39] propose a QuickCheck-like tool for the Agda/Alfa proof assistant, with support for reasoning about the surjectivity (what we call completeness) of generators. They later describe how a user can write a custom generator for a restricted family of inductive datatypes [40]. The associated PhD thesis by Haiyan [58] contains a lot more details and impressive examples of handwritten generators proven surjective, including balanced binary search trees and well-scoped

lambda terms. Just like QuickChick, this line of work leverages its dependently typed setting to allow for strong guarantees about the correctness of generators. Unfortunately, writing such proofs by hand is extremely tedious, to the point of being an entire chapter of a PhD thesis. Staying in the Agda ecosystem, Lindblad later on introduced the Agsy tool [87], with automation support for generating first-order test data. It is one of the first papers to link narrowing and property-based testing, paving the path for the recent random testing advances [31, 43, 81, 82].

Eastlund [41] implemented DoubleCheck, an adaption of QuickCheck for ACL2. Chamarthi *et al.* [27] later proposed a more advanced counterexample finding tool for ACL2s, which uses the full power of the theorem prover and libraries to simplify conjectures so that they are easier to falsify. This work is a great example of how integrating testing and proving together can yield major benefits. While users of proof assistants like Coq, Isabelle or Agda can obtain testing feedback about lemmas or proof goals they have posed themselves, a theorem prover can take this approach to a new level: the proof search algorithm itself can propose helpful lemmas or stronger inductive hypotheses and disprove futile ones using testing. Once again, however, no user customization is allowed for the generators.

Finally, there was an earlier approach to bring random testing in Coq itself by Wilson [126]. They target a relatively small class of testable properties in `Prop`: `True`, `False`, conjuction, disjunction, implication, negation and equality of ground terms, as well as inequalities for natural numbers. They also include a sized automatic generation for simple inductive types; unfortunately, the generation procedure seems to be ill-tuned for types with a branching factor of 3 or more, leading to extremely large data being generated. This is only exacerbated by the lack of support for handwritten generators. Moreover, the tool is run inside Coq, which only allows them to efficiently generate and test very small terms.

7.2 Generating Random Programs

Generating random inputs for testing is a large research area. We focus on the particular sub-area of testing language implementations by generating random *programs*, like we did in our IFC and STLC case studies.

We have already extensively discussed the work of Pałka *et al.* on generating well-typed lambda terms and finding bugs in the GHC strictness analyzer [103, 102]. This work served as inspiration for much of this thesis, as one of the best examples of what a good fine-tuned handcrafted generator can do. There have been other attempts at generating lambda terms. Kennedy and Vytiniotis [73] take the interesting approach of using bit representations of typed programs, and translating large enough random bitstring sequences into typed lambda terms. Yakushev [111] provides enumerators for terms at a given type, using GADTs to do generic programming in Haskell. Their approach allows to obtain typed terms "for free" as the enumeration is done on the constrained GADT space, but it is also, in the author's words, "the slowest". In an interesting more recent take, Tarau [120] used prolog to generate well-typed lambda terms in order to answer statistical queries about such terms. A similar goal is shared among generators for untyped lambda terms [56, 86], where obtaining a reasonable distribution for testing purposes would be even harder.

Klein *et al.* [78] use PLT Redex's QuickCheck-inspired random testing framework to asses the safety of the bytecode verification algorithm for the Racket virtual machine. They observe that naively generated programs only rarely pass bytecode verification (88% discard rate), and that many programs fail verification because of a few common violations that can be easily remedied in a post-generation pass that for instance replaces out-of-bounds indices with random in-bounds ones. These simple changes to the generator are enough for reducing the discard rate (to 42%) and for finding more than two dozen bugs in the virtual machine model, as well as a few in the Racket machine implementation, but three known bugs were missed by this naive generator. The authors conjecture that a more sophisticated test generation technique could probably find these bugs. Adopting advanced random testing techniques, similar to the narrowing concepts in Luck, resulted in much better bug-finding performance [43], ranging from on par with the handwritten generators to an order of magnitude slower for case studies like the one in Pałka *et al.*

CSmith [127] is a C compiler testing tool that generates random C programs, avoiding ones whose behavior is undefined by the C99 standard. When generating

programs, CSmith does not attempt to model the current state of the machine; instead, it chooses program fragments that are correct with respect to some static safety analysis (including type-, pointer-, array-, and initializer-safety, etc.). In our IFC adventures, we found that modeling the actual state of our (much simpler) machine to check that generated programs were hopefully well-formed, as in our generation by execution strategy, made our test-case generation far more effective at finding noninterference bugs. In order to get smaller counterexamples, Regehr *et al.* present C-Reduce [109], a tool for reducing test-case C programs such as those produced by CSmith. They note that conventional shrinking methods usually introduce test cases with undefined behavior; thus, they put a great deal of effort and domain specific knowledge into shrinking well-defined programs only to programs that remain well-defined. To do this, they use a variety of search techniques to find better reduction steps and to couple smaller ones together. Our use of *double shrinking* when testing noninterference is similar to their simultaneous reductions, although we observed no need in our setting for more sophisticated searching methods than the greedy one that is guaranteed to produce a local minimum. Regehr *et al.*'s work on reduction is partly based on Zeller and Hildebrandt's formalization of the delta debugging algorithm *ddmin* [129], a non-domain-specific algorithm for simplifying and isolating failure-inducing program inputs with an extension of binary search. In our experiments, as in Regehr *et al.*'s, domain-specific knowledge was crucial for successful shrinking. In recent work, Koopman *et al.* [80] propose a technique for model-based shrinking.

Another relevant example of testing programs by generating random input is Randoop [100], which generates random sequences of calls to Java APIs. Noting that many generated sequences crash after only a few calls, before any interesting bugs are discovered, Randoop performs *feedback directed* random testing, in which previously found sequences that did not crash are randomly extended. This enables Randoop to generate tests that run much longer before crashing, which are much more effective at revealing bugs. Our generation by execution strategy is similar in spirit, and likewise results in a substantial improvement in bug detection rates.

A state-machine modeling library for (an Erlang version of) QuickCheck has been developed by Quviq [67]. It generates sequences of API calls to a state-

ful system satisfying preconditions formulated in terms of a model of the system state, associating a (model) state transition function with each API call. API call generators also use the model state to avoid generating calls whose preconditions cannot be satisfied. Our generation-by-execution strategy works in a similar way for straightline code.

A powerful and widely used approach to testing is symbolic execution—in particular, *concolic testing* and related dynamic symbolic execution techniques [22, 89]. The idea is to mix symbolic and concrete execution in order to achieve higher code coverage. The choice of which concrete executions to generate is guided by a constraint solver and path conditions obtained from the symbolic executions. Originating with DART [49] and PathCrawler [125], a variety of tools and methods have appeared; some of the state-of-the-art tools include CUTE [117], CREST [19], and KLEE [20] (which evolved from EXE [21]) . We wondered whether dynamic symbolic execution could be used instead of random testing for finding noninterference bugs. As a preliminary experiment, we implemented a simulator for a version of our abstract machine in C and tested it with KLEE. Using KLEE out of the box and without any expert knowledge in the area, we attempted to invalidate various assertions of noninterference. Unfortunately, we were only able to find a counterexample for PUSH*, the simplest possible bug, in addition to a few implementation errors (e.g., out-of-bound pointers for invalid machine configurations). The main problem seems to be that the state space we need to explore is too large [23], so we don't cover enough of it to reach the particular IFC-violating configurations. More recently, Torlak *et al.* [124] used our information-flow stack machine and its bugs with respect to EENI as a case study for their symbolic virtual machine, and report better results.

Balliu *et al.* [8] created ENCOVER, an extension of Java PathFinder, to verify information-flow properties of Java programs by means of concolic testing. In their work, concolic testing is used to extract an abstract model of a program so that security properties can be verified by an SMT solver. While ENCOVER tests the security of individual programs, we use testing to check the soundness of an entire enforcement mechanism. Similarly, Milushev *et al.* [92] have used KLEE for testing the noninterference of individual programs, as opposed to our focus on

testing dynamic IFC mechanisms that are meant to provide noninterference for all programs.

7.3 Dynamic IFC

Even though the focus of Chapter 3 was on the generation aspect of testing noninterference, there is a lot of related work for nonintereference itself. Birgisson *et al.* [14] have a good overview of such related work. Our correct rule for *Store* for the stack machine is called the *no-sensitive-upgrades* policy in the literature and was first proposed by Zdancewic [128] and later adapted to the dynamic IFC setting by Austin *et al.* [3]. To improve precision, Austin *et al.* [4] later introduced a different *permissive-upgrade* policy, where public locations can be written in a high context as long as branching on these locations is later prohibited, and they discuss adding *privatization operations* that would even permit this kind of branching safely. Hedin *et al.* [62] improve the precision of the no-sensitive-upgrades policy by explicit *upgrade annotations*, which raise the level of a location before branching on secrets. They apply their technique to a core calculus of JavaScript that includes objects, higher-order functions, exceptions, and dynamic code evaluation. Birgisson *et al.* [14] show that random testing with QuickCheck can be used to infer upgrade instructions in this setting. The main idea is that whenever a random test causes the program to be stopped by the IFC monitor because it attempts a sensitive upgrade, the program can be rewritten by introducing an upgrade annotation that prevents the upgrade from being deemed sensitive on the next run of the program. In recent work, Bichhawat *et al.* [13] generalize the permissive-upgrade check to arbitrary IFC lattices. They present involved counterexamples, apparently discovered manually while doing proofs. We believe that our testing techniques are well-suited at automatically discovering such counterexamples.

Terauchi *et al.* [121] and later Barthe *et al.* [10] propose a technique for statically verifying the noninterference of individual programs using the idea of self-composition. This reduces the problem of verifying secure information flow for a program P to a safety property for a program $\hat{P}$ derived from P, by composing P with a renaming of itself. Self-composition enables the use of standard (i.e., not

relational [11, 9]) program logics and model checking for showing noninterference. The problem we addressed in Chapter 3 is different: we test the soundness of dynamic IFC mechanisms by randomly generating (a large number of) pairs of related programs. One could imagine extending our technique in the future to testing the soundness of static IFC mechanisms such as type systems [114], relational program logics [11, 9], and self-composition based tools [10].

In recent work Ochoa *et al.* [96] discuss a preliminary model-checking based technique for discovering unwanted information flows in specifications expressed as extended finite state machines. They also discuss about testing systems for unwanted flows using unwinding-based coverage criteria and mutation testing. In a recent position paper, Kinder [74] discusses testing of hyperproperties [34].

7.4 Automatically Generating Constrained Data

The two major research contributions of this thesis (Chapters 4 and 5) both deal with generating constrained random data where the distribution is under user control. Since this work borrows concepts from many different topics in programming languages the potentially related literature is huge. Here, we present just the closest related work.

7.4.1 Random Testing

The works that are most closely related to our own are the narrowing-based approaches of Gligoric *et al.* [48], Claessen *et al.* [31, 32] and Fetscher *et al.* [43]. Gligoric *et al.* use a "delayed choice" approach, which amounts to needed-narrowing, to generate test cases in Java. Claessen *et al.* exploit the laziness of Haskell, combining a narrowing-like technique with FEAT [37], a tool for functional enumeration of algebraic types, to efficiently generate near-uniform random inputs satisfying some precondition. While their use of FEAT allows them to get uniformity by default, it is not clear how user control over the resulting distribution could be achieved. Fetscher *et al.* [43] also use an algorithm that makes local choices with the potential to backtrack in case of failure. Moreover, they add a simple version

of constraint solving, handling equality and disequality constraints. This allows them to achieve excellent performance in testing GHC for bugs (as in [103]). They present two different strategies for making local choices: uniformly at random, or by ordering branches based on their branching factor. While both of these strategies seem reasonable (and somewhat complementary), there is no way of exerting control over the distribution as necessary like we do in Luck. In QuickChick, we build upon their success, adapting narrowing for Coq's inductive relations, showing how to produce Coq generators getting rid of interpretation overheads, and producing proofs of the generators correctness in the process.

7.4.2 Enumeration-Based Testing

An interesting related approach appears in the inspiring work of Bulwahn [18] in the context of Isabelle's [94] QuickCheck [17]. Bulwahn automatically constructs enumerators for a given precondition via a compilation to logic programs using mode inference. This work successfully addresses the issue of generating satisfying valuations for preconditions directly and serves for exhaustive testing of "small" instances, significantly pushing the limit of what is considered "small" compared to previous approaches. Lindblad [87] and Runciman *et al.* [112] also provide support for exhaustive testing using narrowing-based techniques. Instead of implementing mechanisms that resemble narrowing in standard functional languages, Fischer and Kuchen [44] leverage the built-in engine of the functional logic programming language Curry [59] to enumerate tests satisfying a coverage criterion. In a later, black-box approach for Curry, Christiansen and Fischer [30] additionally use *level diagonalization* and randomization to bring larger tests earlier in the enumeration order. While exhaustive testing is useful and has its own merits and advantages over random testing in a lot of domains, we turn to random testing because the complexity of our applications—testing noninterference or optimizing compilers—makes enumeration impractical.

7.4.3 Constraint Solving

Many researchers have turned to constraint-solving-based approaches to generate random inputs satisfying preconditions. In the constraint-solving literature concerning witness generation for SAT problems, the pioneering work of Chakraborty *et al.* [26] stands out because of its efficiency and its guarantees of approximate uniformity. However, there is no way—and no obvious way to add it—of controlling distributions. In addition, their efficiency relies crucially on the *independent support* being small relative to the entire space [1] While true for typical SAT instances, this is not the case for random testing properties, like, for example, noninterference. In fact, a minimal independent support for indistinguishable machines includes one entire machine state and the high parts of another; thus, the benefit from their heuristics may be minimal. Finally, they require logical formulae as inputs, which would require a rather heavy translation from a high-level language like Haskell.

Such a translation from a higher-level language to the logic of a constraint solver has been attempted a few times to support testing [24, 53], the most recent and efficient for Haskell being Target [116]. Target translates preconditions in the form of refinement types, and uses a constraint solver to generate a satisfying valuation for testing. Then it introduces the negation of the generated input to the formula, in order to generate new, different ones. While more efficient than Lazy SmallCheck in a variety of cases, there are still cases where a narrowing-like approach outperforms the tool, further pointing towards the need to combine the two approaches as in Luck. Moreover, the use of an automatic translation and constraint solving does not give any guarantees on the resulting distribution, neither does it allow for user control.

Constraint solving is also used in symbolic-evaluation-based techniques, where the goal is to generate diverse inputs that achieve higher coverage [49, 117, 20, 5, 89, 19, 21, 50, 22]. Recently, in the context of Rosette [124], symbolic execution was used to successfully find bugs in the same information-flow-control case study.

[1] The *support* X of a boolean formula `p` is the set of variables appearing in `p` and the *independent support* is a subset D of X such that no two satisfying assignments for `p` differ only in $X \backslash D$.

7.4.4 Semantics for Narrowing-Based Solvers

Recently, Fowler and Hutton [45] put needed-narrowing-based solvers on a firmer mathematical foundation. They presented an operational semantics of a purely narrowing-based solver, named Reach, proving soundness and completeness. In their concluding remarks, they mention that native representations of primitive datatypes do not fit with the notion of lazy narrowing since they are "large, flat datatypes with strict semantics." In Luck, we were able to exhibit the same behavior for both the primitive integers and their datatype encodings successfully addressing this issue, while at the same time incorporating constraint solving into our formalization.

7.4.5 Probabilistic Programming

Semantics for probabilistic programs share many similarities with the semantics of Luck [91, 51, 52], while the problem of generating satisfying valuations shares similarities with probabilistic sampling [90, 84, 25, 95]. For example, the semantics of the language PROB in the recent probabilistic programming survey of Gordon *et al.* [52] takes the form of probability distributions over valuations, while Luck semantics can be viewed as (sub)probability distributions over constraint sets, which induces a distribution over valuations. Moreover, in probabilistic programs, observations serve a similar role to preconditions in random testing, creating problems for simplistic probabilistic samplers that use *rejection sampling*—i.e., generate and test. Recent advances in this domain, like the work on Microsoft's R2 Markov Chain Monte Carlo sampler [95], have shown promise in providing more efficient sampling, using pre-imaging transformations in analyzing programs. An important difference is in the type of programs usually targeted by such tools. The difficulty in probabilistic programming arises mostly from dealing with a large number of complex observations, modeled by relatively small programs. For example, Microsoft's TrueSkill [63] ranking program is a very small program, powered by millions of observations. In contrast, random testing deals with very complex programs (e.g., a type checker) and a single observation without noise (`observe true`).

We did a simple experiment with R2, using the following probabilistic program

to model indistinguishability of atoms, where we use booleans to model labels:

```
double v1 = Uniform.Sample(0, 10);
double v2 = Uniform.Sample(0, 10);
bool l1 = Bernoulli.Sample(0.5);
bool l2 = Bernoulli.Sample(0.5);
Observer.Observe(l1==l2 && (v1==v2 || l1));
```

Two pairs of doubles and booleans will be indistinguishable if the booleans are equal and, if the booleans are false, so are the doubles. The result was somewhat surprising at first, since all the generated samples have their booleans set to true. However, that is an accurate estimation of the posterior distribution: for every "false" indistinguishable pair there exist 2^{64} "true" ones! Of course, one could probably come up with a better prior or use a tool that allows arbitrary conditioning to skew the distribution appropriately. If, however, for such a trivial example the choices are non-obvious, imagine replacing pairs of doubles and booleans with arbitrary lambda terms, and indistinguishability by a well-typedness relation. Coming up with suitable priors that lead to efficient testing would become an ambitious research problem on its own!

7.4.6 Inductive to Executable Specifications

Finally, at a high level, the algorithm described in Section 5.2 has similarities to earlier attempts at extracting executable specifications from inductive ones [35, 122] in the Coq proof assistant. In principle, we could use the algorithm described in this section to obtain a similar transformation. Consider for example, an inductive predicate `P : A -> B -> C -> Prop`. If we transform it to a predicate `P' : A -> B -> C -> unit -> Prop` by adding `()` as an additional argument at every occurrence of `P`, we could ask our algorithm to generate `x` such that `P' a b c x` holds for all `a`, `b`, and `c`. We would then obtain a partial decision procedure for `P`, based on whether the generator returns `Some` or `None`. In fact, our algorithm can be seen as a generalization of their approach as the derived decision procedures are equivalent (modulo size) for the class of inductive datatypes they handle that yields deterministic functional programs.

7.5 Urns

We conclude the exploration of related work by addressing representations for discrete distributions compared to urns, as well as comparing urns with other self-balancing binary tree structures.

7.5.1 Alternative Discrete Distribution Representations

The literature contains several existing representations for discrete distributions. For example, the QuickCheck [33, 108] and `random-fu` [119] packages both provide support for sampling from such distributions. For each of these approaches, we present in Figure 7.1 its asymptotic run time for initialization and for performing the four operations we want to support, as compared with the run time of an urn for the same operation. The table highlights the trade-offs that we have made: urns cost more to create – their internal structure requires linearithmic time to produce – but achieve competitive sampling performance with cumulative arrays, whereas lists are simple and so benefit from constant-time initialization (`id`) and insertion (`(:)`). Only urns are designed to support the three dynamic update operations, however, and are the only structure to achieve consistent logarithmic performance.

The basic idea behind sampling from a discrete distribution is simple: given the distribution $\{(w_1, x_1), , (w_n, x_n)\}$, we break the range $[0, \sum_{i=1}^{n} w_i)$ into n subranges ($[0, w_1)$, $[w_1, w_1 + w_2)$, etc.) and generate a random number r from the total range; the index of the subrange r belongs to is the index of the desired value. Urns, QuickCheck and `random-fu` follow the same high-level approach, but use different data structure representations.

QuickCheck Perhaps the simplest representation of a discrete distribution over values of type `a` is `[(Weight,a)]` – a list of values paired with their weights. This is the implementation used by QuickCheck's `frequency` combinator and is considered in column 1 of Figure 7.1.[2]

[2]The `frequency` function actually has type `[(Int, Gen a)] -> Gen a`, so it deals with a "distribution over distributions"; however, all the representations function equally well holding `a`s and `Gen a`s, so we elide this extra detail.

Operation	Lists	Arrays	Urns
Create from list	$O(1)$	$O(n)$	$O(n)$
Sample	$O(n)$	$O(\log n)$	$O(\log n)$
Total weight	$O(n)$	$O(1)$	$O(1)$
Insert	$O(1)$	$O(n)$	$O(\log n)$
Remove	$O(n)$	$O(n)$	$O(\log n)$
Update/Replace	$O(n)$	$O(n)$	$O(\log n)$

Figure 7.1: Comparison of runtimes for operations on different distribution data structures; n is the number of values in distribution.

Using a list representation is very simple, uses only standard data types, and requires only simple, local invariants on the input data: that the list is nonempty and that its weights aren't all 0. However, while this simple implementation works for small cases it has some obvious inefficiencies: recalculation of the total weight and worst-case linear traversal to generate samples.

On the plus side, inserting a new value into the distribution is constant-time: if we want to add the new value `x'` with weight `w'` to the distribution `d`, we can simply cons them onto the front to produce the new distribution `(w',x'):d`. Because all the invariants are local, no other computation is needed. Other modifications – deleting a value, replacing a value, or updating the weight of a value – take linear time in the worst case, however, as modifying the structure of a linked list always does.

`random-fu` The `random-fu` package [119] uses a similar representation for discrete distributions – also called categorical distributions – in its `Data.Random.Distribution.Categorical.Categorical` type. However, it instead uses an array (specifically, a `Vector` from the `vectory` package [61]) and pairs values with their *cumulative* weights: the distribution

$$\{(w_0, x_0), (w_1, x_1), , (x_n, x_n)\}$$

becomes the array

```
[(w0,x0), (w0+w1,x1), ..., (w0+w1+...+wn,xn)]
```

This representation is considered in column 2 of Fig. 7.1. Now, the total weight is stored in the last position in the vector, which is accessible in constant time; and we can use binary search to find which bucket an element of $[0, \texttt{total})$ belongs to in logarithmic time. This works because each position in the vector stores the upper bound on its bucket, which is the value that the index needs to be compared against.

As it is not a design goal of the library, `random-fu` itself does not expose any operations for dynamic updates. Nevertheless, we can consider how this representation would work if were to extend it to support them. As it happens, this representation requires linear time for all updates. For delete, update, and replace, this is independent of the runtime of the array operations; since the array stores *cumulative* weights, modifying any weight in the middle of the array requires modifying all subsequent weights as well. To insert an value, we can add a new value to the end of the array without the need to update any other weights; however, because our arrays are immutable, this still requires copying the entire array and thus takes linear time. If our distribution were mutable (in `ST` or `IO`), we could get amortized constant-time append, and thus improve the efficiency of insertion.

7.5.2 Balancing Binary Trees

Urns are reminiscent of other data structures based on complete binary trees: certain variants of heaps store a tree linearized into an array, with the children of node i at indices $2i + 1$ and $2i + 2$ (using 0-indexing). If we always fill the array from left to right – taking care to convert leaves into nodes when necessary – then we will always have a balanced tree. However, as with all array-based representations, updates in a purely functional setting require copying the entire array, and so cost $O(n)$; we get no sharing at all. Urns provide an elegant, purely functional alternative, filling a very real need in the random testing community as we saw in the previous section. Additionally, since urns are immutable trees with no important laziness properties, they may be used in a persistent context with the same performance as when used ephemerally. That is to say, backtracking to

a previous state of an urn costs nothing.

Even in the functional setting, there already exist self-balancing data structures like red-black trees and AVL trees. However, these data structures maintain complicated invariants, and are notoriously difficult to get (and prove) correct [29]. Moreover, QuickCheck generators, one of the main applications of urns, cannot be given an `Ord` structure: they are implemented as Haskell functions. Since urns do not maintain a specific arrangement of their values, they can contain generators, as well as arbitrary functions, `IO` actions, or other objects without imposing any constraints.

Chapter 8

Conclusion and Future Work

In this thesis we described work that aims to amplify both the efficiency and applicability of random testing, especially in the context of programming language design and verification. We introduced QuickChick, the first complete property-based random testing tool for Coq. We developed Luck, a domain-specific language for writing generators as lightly annotated predicates, where disparate techniques from the literature (narrowing and constraint solving) are synergistically combined under user control. We also ported to QuickChick ideas formalized and evaluated in Luck, further facilitating future verification efforts.

This thesis was largely focused on *generation*. The two major research papers [81, 82] that originated from this work both address the problem of automatically deriving generators for well-distributed random data satisfying a certain predicate. The main artifact produced, QuickChick, brings these advances in the popular Coq proof assistant: the first step towards bringing the benefits of random testing in Coq. However, there is a lot of potential for future work, both in the context of QuickChick and more generally.

First, the automatically derived generators we presented, while very efficient for a large class of interesting preconditions, still suffer from the standard narrowing drawback: particular preconditions can force generation of variables early, potentially leading to a lot of backtracking. This problem is compounded by our choice to compile inductive predicates to actual generators in the host language.

Addressing this issue is a significant research challenge that I aim to address in the future.

Moreover, a latent issue in this thesis is how to properly *evaluate* random generators. In our case we focused on two case studies, information-flow control experiments and the simply typed lambda calculus, where we encountered a particularly fortunate situation: we could systematically introduce mutations where every mutant that was *not* discovered by our testing revealed something important about our testing, artifact or property, completely eliminating the "equivalent mutant problem" [71]. This problem is not unique to property-based random testing: establishing a proper way of evaluating research for testing tools and methodologies is one of my short-term goals.

In addition, while the work described in this thesis makes significant progress in the context of generators, there are other aspects of random testing that are still lacking in QuickChick. In particular, a lot of manual effort is currently necessary to translate between inductively defined predicates and executable decidability procedures that can be used to actually check whether the predicates hold, a problem glossed over in Chapter 5. The algorithm described in that chapter should be generalizable to derive such procedures from a given inductive predicate.

Shrinking is the other aspect of random testing that could use a more thorough treatment. While recent work, including this one, has set the foundation of what it means for a generator to be correct, there is no similar treatment for shrinkers. Indeed, other than the idea that shrinking should not produce cycles to avoid nontermination, there is, to my knowledge, no explicit correctness criterion. Moreover, just like generate-and-test can be unacceptably inefficient for certain preconditions, a shrink-and-test approach to shrinking data satisfying preconditions can also incur large overheads. Even though this problem is slightly less impactful than the generator one, attempting to derive shrinkers from the structure of a predicate is a necessary step towards better random testing.

Finally, another interesting direction for future work is exploring the applicability of probabilistic programming in random testing. The two areas share many similarities and it should be really interesting to see how probabilistic sampling techniques fare against custom crafted generators for complex artifacts.

Appendix A

Core Luck Proofs

Before we reach the other main theorems we need to prove preservation for the narrowing semantics; to do that we first need to prove that the typing map of constraint sets only increases when narrowing.

Lemma A.0.0.1 (Narrowing Effect on Type Environments)**.**

$$e \dashv \kappa \Downarrow_q^t \kappa' \vDash v \;\Rightarrow\; U(\kappa')|_{U(\kappa)} \equiv U(\kappa)$$

Proof: By induction on the derivation.
Case N-Base: $U(\kappa)|_{U(\kappa)} \equiv U(\kappa)$ by the definition of restriction.
Case N-Pair: By the inductive hypothesis we have $U(\kappa_1)|_{U(\kappa)} \equiv U(\kappa)$ and $U(\kappa_2)|_{U(\kappa_1)} \equiv U(\kappa_1)$. The result follows by transitivity.
Case N-CasePair-P: Similar to **N-Pair**.
Case N-CasePair-U: By the inductive hypothesis we have $U(\kappa_a)|_{U(\kappa)} \equiv U(\kappa)$ and $U(\kappa')|_{U(\kappa_c)} \equiv U(\kappa_c)$. By transitivity, we only need to show that $U(\kappa_c)|_{U(\kappa_a)} \equiv U(\kappa_a)$. This follows by transitivity of restrict (through $U(\kappa_b)$), and the specifications of *fresh* and *unify*.
Cases N-L, N-R: The induction hypothesis gives us $U(\kappa')|_{U(\kappa)} \equiv U(\kappa)$, which is exactly what we want to prove.
Cases N-Case-L, N-Case-R, N-App: Similar to **N-CasePair-P**.
Cases N-Case-U-*: For each of the four cases derived by inlining *choose*, we

proceed exactly like **N-CasePair-U**.

Cases N-Narrow-*: For each of the four cases derived by inlining *choose* the result follows as in **N-CasePair-U**, with additional uses of transitivity to accommodate the narrowing derivations for e_1 and e_2.

Case N-Bang: Directly from the induction hypothesis, as in **N-L**.

□

We also need to prove a form of context invariance for unknowns: we can substitute a typing map U with a supermap U' in a typing relation.

Lemma A.0.0.2 (Unknown Invariance)**.**

$$\left.\begin{array}{l} \Gamma; U \vdash e : T \\ U'|_U \equiv U \end{array}\right\} \Rightarrow\ \Gamma; U' \vdash e : T$$

Proof: By induction on the typing derivation for e. The only interesting case is the one regarding unknowns: we know for some unknown u that $U(u) = T$ and that $U'|_U \equiv U$ and want to prove that $\Gamma; U(U') \vdash u : T$. By the **T-U** rule we just need to show that $U'(u) = T$, which follows the definition of $\equiv$ and $|.$ for maps. □

We can now prove preservation: if a constraint set κ is well typed and an expression e has type T in $U(\kappa)$ and the empty context, then if we narrow $e \dashv \kappa$ to obtain $\kappa' \vDash v$, κ' will be well typed and v will also have the same type T in $U(\kappa')$.

Theorem A.0.0.3 (Preservation)**.**

$$\left.\begin{array}{l} e \dashv \kappa \Downarrow_q^t \kappa' \vDash v \\ U(\kappa) \vdash e : T \\ \vdash \kappa \end{array}\right\} \Rightarrow \left\{\begin{array}{l} U(\kappa') \vdash v : T \\ \vdash \kappa' \end{array}\right.$$

Proof: Again, we proceed by induction on the narrowing derivation.

Case N-Base: Since $v = e$ and $\kappa' = \kappa$, the result follows immediately from the hypothesis.

Case N-Pair: We have

$$e_1 \dashv \kappa \Downarrow_{q_1}^{t_1} \kappa_1 \vDash v_1 \ \text{ and } \ e_2 \dashv \kappa_1 \Downarrow_{q_2}^{t_2} \kappa_2 \vDash v_2.$$

The inductive hypothesis for the first derivation gives us that

$$\forall T'.\ \left.\begin{array}{l} U(\kappa) \vdash e_1 : T' \\ \vdash \kappa \end{array}\right\} \Rightarrow \left\{\begin{array}{l} U(\kappa_1) \vdash v_1 : T' \\ \vdash \kappa_1 \end{array}\right. .$$

Similarly, the second inductive hypothesis gives us that

$$\forall T'.\ \left.\begin{array}{l} U(\kappa_1) \vdash e_2 : T' \\ \vdash \kappa_1 \end{array}\right\} \Rightarrow \left\{\begin{array}{l} U(\kappa_2) \vdash v_2 : T' \\ \vdash \kappa_2 \end{array}\right. .$$

The typing assumption of the theorem states that

$$U(\kappa) \vdash (e_1, e_2) : T.$$

We want to show that

$$U(\kappa_2) \vdash (v_1, v_2) : T \ \text{ and } \ \vdash \kappa_2.$$

By inversion on the typing relation for (e_1, e_2) we know that there exist T_1, T_2 such that

$$T = T_1 \times T_2 \ \text{ and } \ U(\kappa) \vdash e_1 : T_1 \ \text{ and } \ U(\kappa) \vdash e_2 : T_2.$$

We first instantiate the first inductive hypothesis on T_1 which gives us $U(\kappa_1) \vdash v_1 : T_1$ and $\vdash \kappa_1$. Then, to instantiate the second one on T_2 and obtain $U(\kappa_2) \vdash v_2 : T_2$ and $\vdash \kappa_2$, we need to show that $U(\kappa_1) \vdash e_2 : T_2$, which follows by combining Lemma A.0.0.1 and Unknown Invariance (Lemma A.0.0.2). The same combination also gives us $U(\kappa_2) \vdash e_1 : T_1$. The result follows by the **T-Pair** constructor.

Case N-CasePair-P: We have

$$e \dashv \kappa \Downarrow_q^t \kappa_a \vDash (v_1, v_2) \ \text{ and } \ e'[v_1/x,\, v_2/y] \dashv \kappa_a \Downarrow_{q'}^{t'} \kappa' \vDash v.$$

By the inductive hypothesis,

$$\forall T'.\ \left.\begin{array}{l} U(\kappa) \vdash e : T' \\ \vdash \kappa \end{array}\right\} \Rightarrow \left\{\begin{array}{l} U(\kappa_a) \vdash (v_1, v_2) : T' \\ \vdash \kappa_a \end{array}\right.$$

and

$$\forall T'.\ \left.\begin{array}{l} U(\kappa_a) \vdash e'[v_1/x, v_2/y] : T' \\ \vdash \kappa_a \end{array}\right\} \Rightarrow \left\{\begin{array}{l} U(\kappa') \vdash v : T' \\ \vdash \kappa' \end{array}\right. .$$

The typing assumption gives us

$$U(\kappa) \vdash \mathit{case}\ e\ \mathit{of}\ (x, y) \to e' : T.$$

Inversion on this typing relation means that there exist types T_1, T_2 such that

$$U(\kappa) \vdash e : T_1 \times T_2 \ \text{ and } \ (y \mapsto T_2, x \mapsto T_1); U(\kappa) \vdash e' : T$$

We want to prove that

$$U(\kappa') \vdash v : T \ \text{ and } \ \vdash \kappa'.$$

We instantiate the first inductive hypothesis on $T_1 \times T_2$ and use inversion on the resulting typing judgment for (v_1, v_2), which yields

$$\vdash \kappa_a \ \text{ and } \ U(\kappa_a) \vdash v_1 : T_1 \ \text{ and } \ U(\kappa_a) \vdash v_2 : T_2.$$

By the second inductive hypothesis, we only need to show that

$$\varnothing; U(\kappa_a) \vdash e'[v_1/x, v_2/y] : T \ \wedge \ \vdash \kappa_a$$

Applying the Substitution Lemma twice, Unknown Invariance and Lemma A.0.0.1 yields the desired result.

Case N-CasePair-U: Similarly to the previous case, we have

$$e \dashv \kappa \Downarrow_q^t \kappa_a \vDash u \ \text{ and } \ e'[u_1/x,\ u_2/y] \dashv \kappa_c \Downarrow_{q'}^{t'} \kappa' \vDash v,$$

where $(\kappa_b, [u_1, u_2]) = \mathit{fresh}\ \kappa_a\ [\overline{T}_1, \overline{T}_2]$
and $\kappa_c = \mathit{unify}\ \kappa_b\ (u_1, u_2)\ u$ and $U(\kappa) \vdash e : \overline{T}_1 \times \overline{T}_2$.

As in the previous case we have two inductive hypotheses

$$\forall T'.\ \left.\begin{array}{l} U(\kappa) \vdash e : T' \\ \vdash \kappa \end{array}\right\} \Rightarrow \left\{\begin{array}{l} U(\kappa_a) \vdash u : T' \\ \vdash \kappa_a \end{array}\right.$$

and

$$\forall T'.\ \left.\begin{array}{l} U(\kappa_c) \vdash e'[u_1/x, u_2/y] : T' \\ \vdash \kappa_c \end{array}\right\} \Rightarrow \left\{\begin{array}{l} U(\kappa') \vdash v : T' \\ \vdash \kappa' \end{array}\right. .$$

The same typing assumption,

$$U(\kappa) \vdash \mathit{case}\ e\ \mathit{of}\ (x, y) \to e' : T,$$

can be inverted to introduce T_1 and T_2 such that

$$U(\kappa) \vdash e : T_1 \times T_2 \ \text{ and } \ (y \mapsto T_2, x \mapsto T_1); U(\kappa) \vdash e' : T.$$

By type uniqueness, $\overline{T}_1 = T_1$ and $\overline{T}_2 = T_2$. Once again, we want to prove that

$$U(\kappa') \vdash v : T \ \text{ and } \ \vdash \kappa'.$$

Like in the previous case, by the first inductive hypothesis instantiated on $T_1 \times T_2$ we get that κ_a is well typed and u has type $T_1 \times T_2$ in κ_a. By the specification of *fresh* (Lemma 4.3.1.4) we get that κ_b is well typed and that $U(\kappa_b) = U(\kappa_a) \oplus u_1 \mapsto T_1 \oplus u_2 \mapsto T_2$, where u_1 and u_2 do not appear in $U(\kappa_a)$. By the specification of *unify* we know that $U(\kappa_c) = U(\kappa_b)$ and Lemma 4.3.1.9 means that κ_c is well typed. Finally, we instantiate the second inductive hypothesis on T, using the Substitution Lemma and Unknown Invariance to prove its premise.

Cases N-L, N-R, N-Fold: Follows directly from the induction hypothesis after inversion of the typing derivation for e.

Cases N-Case-L, N-Case-R, N-Unfold-F: Similar to **N-CasePair-P**.

Cases N-Unfold-U, N-Case-U-*: The unknown case for unfold as well as the four cases derived by inlining *choose* are similar to **N-CasePair-U**.

Case N-App: We have

$$e_0 \dashv \kappa \Downarrow_{q_0}^{t_0} \kappa_a \vDash v_0, \quad e_1 \dashv \kappa_a \Downarrow_{q_1}^{t_1} \kappa_b \vDash v_1,$$
$$e_2[v_0/f,\, v_1/x] \dashv \kappa_b \Downarrow_{q_2}^{t_2} \kappa' \vDash v,$$

as well as the corresponding inductive hypotheses, where v_0 is of the form $(rec\ (f : T_1 \to T_2)\ x = e_2)$.

The preservation typing assumption states that κ is well typed and that $U(\kappa) \vdash (e_0\ e_1) : T_2'$. Inverting this typing relation gives us that

$$U(\kappa) \vdash e_0 : T_1' \to T_2' \text{ and } U(\kappa) \vdash e_1 : T_1'.$$

By the first inductive hypothesis,

$$\forall T'.\ \left.\begin{array}{l} U(\kappa) \vdash e_0 : T' \\ \vdash \kappa \end{array}\right\} \Rightarrow \left\{\begin{array}{l} U(\kappa_a) \vdash v_0 : T' \\ \vdash \kappa_a \end{array}\right. .$$

Instantiating this hypothesis on $T_1' \to T_2'$ and inverting the resulting typing relation gives us that $T_1' = T_1$ and $T_2' = T_2$. The remainder of the proof is similar to the second part of **N-CasePair-P**.

Case N-Bang: We have

$$e \dashv \kappa \Downarrow_q^t \kappa_a \vDash v \text{ and } sampleV\ \kappa_a\ v \Rightarrow_{q'}^{t'} \kappa'.$$

By the inductive hypothesis,

$$\forall T'.\ \left.\begin{array}{l} U(\kappa) \vdash e : T' \\ \vdash \kappa \end{array}\right\} \Rightarrow \left\{\begin{array}{l} U(\kappa_a) \vdash v : T' \\ \vdash \kappa_a \end{array}\right. .$$

The typing premise of preservation states that e has type $\overline{T}$ in $U(\kappa)$, and we can instantiate the inductive hypothesis with $\overline{T}$. By the specification of *sample*, $U(\kappa') = U(\kappa)$ and the typing lemma for *sample* yields that κ' is well typed which concludes the proof.

Case N-Inst-*:

Each of the four cases derived by inlining *choose* are similar. The typing premise is $\varnothing; U(\kappa) \vdash e \leftarrow (e_1, e_2) : T$, while we know that $U(\kappa) \vdash e : \overline{T}_1 \times \overline{T}_2$. Inverting the premise and using type uniqueness allows us to equate T with $\overline{T}_1 \times \overline{T}_2$ and also gives us that e_1 and e_2 have type *nat* in κ.

We have that $e \rightleftharpoons \kappa \Downarrow_q^t \kappa_a \vDash v$ and the corresponding inductive hypothesis:

$$\forall T'.\ \left.\begin{array}{l} U(\kappa) \vdash e : T' \\ \vdash \kappa \end{array}\right\} \Rightarrow \left\{\begin{array}{l} U(\kappa_a) \vdash v : T' \\ \vdash \kappa_a \end{array}\right. .$$

We instantiate it to $\overline{T}_1 \times \overline{T}_2$. Using the narrowing of types and Unknown Invariance, we get that e_1 and e_2 have type *nat* in κ_a.

We proceed similarly for the derivations

$$e_1 \rightleftharpoons \kappa_a \Downarrow_{q_1}^{t_1} \kappa_b \vDash v_1 \ \text{ and } \ e_2 \rightleftharpoons \kappa_b \Downarrow_{q_2}^{t_2} \kappa_c \vDash v_2,$$

propagating type information using Lemma A.0.0.1, using the induction hypothesis and Unknown Invariance, to obtain that κ_c is well typed, v has type $\overline{T}_1 \times \overline{T}_2$, and v_1 and v_2 have type *nat*.

Continuing with the flow of the rule,

$$\mathit{sampleV}\ \kappa_c\ v_1 \Rightarrow_{q_1'}^{t_1'} \kappa_d \ \text{ and } \ \mathit{sampleV}\ \kappa_d\ v_2 \Rightarrow_{q_2'}^{t_2'} \kappa_e,$$

and the specification for *sample* lifted to *sampleV* yield $U(\kappa_e) = U(\kappa_d) = U(\kappa_c)$ and κ_e is well typed.

We can then apply the specification of *fresh* to the generation of the unknowns u_1 and u_2,

$$(\kappa_0, [u_1, u_2]) = \mathit{fresh}\ \kappa_e\ [\overline{T}_1, \overline{T}_2],$$

which means $U(\kappa_0) = U(\kappa_e) \oplus u_1 \mapsto \overline{T}_1 \oplus u_2 \mapsto \overline{T}_2$ and $\vdash \kappa_0$. Therefore, $U(\kappa_0)|_{U(\kappa_e)} \equiv U(\kappa_e)$ and by Unknown Invariance the type of v carries over to $U(\kappa_0)$. In addition, $L_{\overline{T}_1+\overline{T}_2}\ u_1$ and $R_{\overline{T}_1+\overline{T}_2}\ u_2$ have the same type as v in κ_0.

The two unifications operate on κ_0,

$$\kappa_l = \mathit{unify}\ \kappa_0\ v\ (L_{\overline{T}_1+\overline{T}_2}\ u_1)$$

and

$$\kappa_r = \mathit{unify}\ \kappa_0\ v\ (R_{\overline{T}_1+\overline{T}_2}\ u_2),$$

and the specification of unify applies to give us that $U(\kappa_0) = U(\kappa_l) = U(\kappa_r)$, as well as $\vdash\ \kappa_l$ and $\vdash\ \kappa_r$. Thus, for both κ_l and κ_r, v has type $\overline{T}_1 \times \overline{T}_2$. Since *choose* will pick one of κ_l or κ_r to return, this concludes the proof.

With preservation for the narrowing semantics proved, we only need one very simple lemma about the interaction between variable and valuation substitution in expressions:

Lemma A.0.0.4 (Substitution Interaction)**.**

$$\left.\begin{array}{l}\sigma(e) = e' \\ \sigma(v) = v'\end{array}\right\} \Rightarrow \sigma(e[v/x]) = e'[v'/x]$$

Proof: The result follows by induction on $\sigma(e) = e'$ and case splitting on whether x is equal to any variable encountered. □

Before we formally state and prove soundness, we need two technical lemmas about unknown inclusion in domains.

Lemma A.0.0.5 (Narrow Result Domain Inclusion)**.**

$$\left.\begin{array}{l}e \dashv \kappa \Downarrow_q^t \kappa' \vDash v \\ (\forall u.\ u \in e \Rightarrow u \in \mathit{dom}(\kappa))\end{array}\right\} \Rightarrow\ \forall u.\ u \in v \Rightarrow u \in \mathit{dom}(\kappa')$$

Proof: Straightforward induction on the narrowing derivation. □

Lemma A.0.0.6 (Narrow Increases Domain)**.**

$$\left.\begin{array}{l}e \dashv \kappa \Downarrow_q^t \kappa' \vDash v \\ u \in \mathit{dom}(\kappa)\end{array}\right\} \Rightarrow\ u \in \mathit{dom}(\kappa')$$

Proof: Straightforward induction on the narrowing derivation, using the specifications of *fresh*, *unify* and *sample*. □

We also need a sort of inverse to Substitution Interaction:

Lemma A.0.0.7 (Inverse Substitution Interaction)**.**

$$\left.\begin{array}{c}\sigma(e_2) = e_2' \\ \sigma(e_1[e_2/x]) = e_1''\end{array}\right\} \Rightarrow\ \exists e_1'.\ \sigma(e_1) = e_1'$$

Proof: By induction on e_1, inversion of the substitution relation and case analysis on variable equality when necessary. □

Theorem A.0.0.8 (Soundness)**.**

$$\left.\begin{array}{l} e \dashv \kappa \Downarrow_t^q \kappa' \vDash v \\ \sigma'(v) = v_p\ \wedge\ \sigma' \in [\![\kappa']\!] \\ \forall u.\ u \in e \Rightarrow u \in dom(\kappa)\end{array}\right\} \Rightarrow\ \exists \sigma\ e_p. \left\{\begin{array}{l}\sigma'|_\sigma \equiv \sigma \\ \sigma \in [\![\kappa]\!] \\ \sigma(e) = e_p \\ e_p \Downarrow v_p\end{array}\right.$$

Proof: By induction on the narrowing derivation.

Case N-Base: In the base case $e = v$ and therefore the soundness witnesses are trivially σ' and v_p.

Case N-Pair: We know that

$$e_1 \dashv \kappa \Downarrow_{q_1}^{t_1} \kappa_1 \vDash v_1 \text{ and } e_2 \dashv \kappa_1 \Downarrow_{q_2}^{t_2} \kappa' \vDash v_2.$$

By inversion of the substitution $\sigma'(v)$ we know that $v = (v_1, v_2)$, $\sigma(v_1) = v_{p_1}$ and $\sigma(v_2) = v_{p_2}$.

By the induction hypothesis for e_{p_2}, we get that

$$\left.\begin{array}{l}\sigma'(v_2) = v_{p_2}\ \wedge\ \sigma' \in [\![\kappa']\!] \\ \forall u.u \in e_2 \Rightarrow u \in dom(\kappa_1)\end{array}\right\} \Rightarrow\ \exists \sigma_1\ e_{p_2}. \left\{\begin{array}{l}\sigma'|_{\sigma_1} \equiv \sigma_1 \\ \sigma_1 \in [\![\kappa_1]\!] \\ e_{p_2} \Downarrow v_{p_2} \\ \sigma_1(e_2) = e_{p_2}\end{array}\right. .$$

Its only premise that is not an assumption can be discharged using the lemma about narrowing increasing domain (Lemma A.0.0.6).

The induction hypothesis for e_{p_1} states that

$$\left.\begin{array}{l}\sigma_1(v_1) = v_{p_1} \ \wedge \ \sigma_1 \in [\![\kappa_1]\!] \\ \forall u.u \in e_1 \Rightarrow u \in dom(\kappa)\end{array}\right\} \Rightarrow \exists \sigma \ e_{p_1}. \left\{\begin{array}{l}\sigma_1|_\sigma \equiv \sigma \\ \sigma \in [\![\kappa]\!] \\ e_{p_1} \Downarrow v_{p_1} \\ \sigma(e_1) = e_{p_1}\end{array}\right. .$$

Proving that $\sigma_1(v_1) = v_{p_1}$ is where the requirement that unknowns be bound in the input κ comes into play: since σ_1 is a restriction of σ', they assign the same values to all unknowns that σ_1 assigns a value to. Using Lemma A.0.0.5, we can show that all unknowns in v_1 are included in the domain of κ_1 and therefore $\sigma_1(v_1) = \sigma'(v_1) = v_{p_1}$.

The final witnesses for the **N-Pair** case are σ and (e_{p_1}, e_{p_2}). The conclusion follows using the transitivity of restrict and the **P-Pair** constructor.

Case N-CasePair-P: We know that

$$e \dashv \kappa \Downarrow_q^t \kappa_a \vDash (v_1, v_2) \text{ and } e'' \dashv \kappa_a \Downarrow_{q'}^{t'} \kappa' \vDash v,$$

where $e'' = e'[v_1/x, v_2/y]$. The inductive hypothesis for the second derivation gives us

$$\left.\begin{array}{l}\sigma'(v) = v_p \ \wedge \ \sigma' \in [\![\kappa']\!] \\ \forall u.u \in e'' \Rightarrow u \in dom(\kappa_a)\end{array}\right\} \Rightarrow \exists \sigma_1 \ e''_p. \left\{\begin{array}{l}\sigma'|_{\sigma_1} \equiv \sigma_1 \\ \sigma_1 \in [\![\kappa_a]\!] \\ e''_p \Downarrow v_p \\ \sigma_1(e'') = e''_p\end{array}\right. .$$

After discharging the premises using Lemma A.0.0.5 and Lemma A.0.0.6, we can investigate the shape of the e''_p witness. First note that, because of the domain inclusions, there exist v_{p_1}, v_{p_2} such that $\sigma_1(v_1) = v_{p_1}$ and $\sigma_1(v_2) = v_{p_2}$. But then, by applying the Inverse Substitution Interaction lemma (Lemma A.0.0.7) we know that there exists e'_p such that $\sigma_1(e') = e_p$.

By the inductive hypothesis for e, we get that

$$\left.\begin{array}{l}\sigma_1((v_1, v_2)) = (v_{p_1}, v_{p_2}) \\ \sigma_1 \in [\![\kappa_a]\!] \\ \forall u.u \in e \Rightarrow u \in dom(\kappa)\end{array}\right\} \Rightarrow \exists \sigma\ e_p. \left\{\begin{array}{l}\sigma_1|_\sigma \equiv \sigma \\ \sigma \in [\![\kappa]\!] \\ e_p \Downarrow (v_{p_1}, v_{p_2}) \\ \sigma(e) = e_p\end{array}\right. .$$

This allows us to provide witnesses for the entire case: σ and *case* e_p *of* $(x, y) \to e'_p$. Using the transitivity of restrict and the **P-CasePair** rule, we only need to show that $e''_p = e'_p[v_{p_1}/x, v_{p_2}/y]$, which follows from two applications of the (normal) Substitution Interaction lemma (Lemma A.0.0.4).

Case N-CasePair-U: This case is largely similar with **N-CasePair-P**, with added details for dealing with *fresh* and *unify*. We have

$$e \dashv \kappa \Downarrow^t_q \kappa_a \vDash u' \text{ and } e'' \dashv \kappa_c \Downarrow^{t'}_{q'} \kappa' \vDash v,$$

where

$$\begin{gathered} e'' = e'[u_1/x,\ u_2/y], \\ (\kappa_b, [u_1, u_2]) = \mathit{fresh}\ \kappa_a\ [\overline{T}_1, \overline{T}_2], \\ \kappa_c = \mathit{unify}\ \kappa_b\ (u_1, u_2)\ u'. \end{gathered}$$

By the inductive hypothesis for the second derivation, we get

$$\left.\begin{array}{l}\sigma'(v) = v_p\ \wedge\ \sigma' \in [\![\kappa']\!] \\ \forall u.u \in e'' \Rightarrow u \in dom(\kappa_c)\end{array}\right\} \Rightarrow \exists \sigma_1\ e''_p. \left\{\begin{array}{l}\sigma'|_{\sigma_1} \equiv \sigma_1 \\ \sigma_1 \in [\![\kappa_c]\!] \\ e''_p \Downarrow v_p \\ \sigma_1(e'') = e''_p\end{array}\right. .$$

Discharging the inclusion premise is slightly less trivial in this case, since it requires using the specifications of fresh and unify to hand the case where $u = u_1$ or $u = u_2$.

Then, by the definition of κ_c, we know that σ_1 is in the denotation of *unify* κ_b (u_1, u_2) u'. But, by the specification of unify, $\sigma_1|_{dom(\kappa_b)} \in [\![\kappa_b]\!]$. Since *fresh* preserves the domains of constraints sets, that also means that $\sigma_1|_{dom(\kappa_a)} \in [\![\kappa_a]\!]$. In the

following, let $\sigma_1' = \sigma_1|_{dom(\kappa_a)}$; then, since $u \in dom(\kappa_a)$ by Lemma A.0.0.5, there exists some value v_u such that $\sigma_1'(u) = v_u$.

We can now use the inductive hypothesis for the first narrowing derivation that states that:

$$\left.\begin{array}{l}\sigma_1'(u) = v_u \ \wedge \ \sigma_1' \in [\![\kappa_a]\!] \\ \forall u.u \in e \Rightarrow u \in dom(\kappa)\end{array}\right\} \Rightarrow \ \exists \sigma \ e_p. \left\{\begin{array}{l}\sigma_1'|_\sigma \equiv \sigma \\ \sigma \in [\![\kappa]\!] \\ e_p \Downarrow v_u \\ \sigma(e) = e_p\end{array}\right. .$$

We conclude as in **N-CasePair-P**, with σ and *case* e_p *of* $(x, y) \rightarrow e_p'$ as the witnesses, where e_p' is obtained as in the previous case by investigating the shape of e_p''.

Cases N-L, N-R, N-Fold: These cases follow similarly to **N-Pair**.

Cases N-Case-L, N-Case-R, N-Unfold-F, N-After: These cases follow similarly to **N-CasePair-P**.

Cases N-Case-U, N-Unfold-U: These cases follow similarly to **N-CasePair-U**.

Case N-App: We have

$$e_0 \dashv \kappa \Downarrow_q^t \kappa_a \vDash v_0, \quad e_1 \dashv \kappa_a \Downarrow_q^t \kappa_b \vDash v_1,$$
$$e_2' \dashv \kappa_b \Downarrow_{q'}^{t'} \kappa' \vDash v,$$

where v_0 is of the form $(rec\ (f : T_1 \rightarrow T_2)\ x = e_2)$ and $e_2' = e_2[v_0/f,\ v_1/x]$.

By the inductive hypothesis for the third derivation we get that

$$\left.\begin{array}{l}\sigma'(v) = v_p \ \wedge \ \sigma' \in [\![\kappa']\!] \\ \forall u.u \in e_2' \Rightarrow u \in dom(\kappa_b)\end{array}\right\} \Rightarrow \ \exists \sigma_2 \ e_{p_2}'. \left\{\begin{array}{l}\sigma'|_{\sigma_2} \equiv \sigma_2 \\ \sigma_2 \in [\![\kappa_b]\!] \\ e_{p_2}' \Downarrow v_p \\ \sigma_2(e_2') = e_{p_2}'\end{array}\right. .$$

As in **N-CasePair-P**, we can prove that there exists v_{p_0} and v_{p_1} such that $\sigma_2(v_0) = v_{p_0}$ and $\sigma_2(v_1) = v_{p_1}$.

By the inductive hypothesis for the evaluation of the argument we get that there exist σ_1 and e_{p_1} such that $\sigma_2|_{\sigma_1} \equiv \sigma_1$ and $\sigma_1 \in [\![\kappa_a]\!]$ and $\sigma(e_1) = e_{p_1}$ and $e_{p_1} \Downarrow v_{p_1}$.

Since σ_1 is a restriction of σ_2 and because of the inclusion hypotheses, σ_1 also maps the lambda to v_{p_0}, which allows us to use the last inductive hypothesis:

$$\left.\begin{array}{l}\sigma_1(v_0) = v_{p_0} \ \wedge \ \sigma_1 \in [\![\kappa_a]\!] \\ \forall u.u \in e_0 \Rightarrow u \in dom(\kappa)\end{array}\right\} \Rightarrow \ \exists \sigma \ e_{p_0}. \left\{\begin{array}{l}\sigma_1|_\sigma \equiv \sigma \\ \sigma \in [\![\kappa]\!] \\ e_{p_0} \Downarrow v_{p_0} \\ \sigma(e_0) = e_{p_0}\end{array}\right. .$$

By inverting the substitution for σ_1 in the lambda expression, we know that there exists e_{p_2}, such that

$$v_{p_0} = (rec \ (f : T_1 \to T_2) \ x = e_{p_2}) \ \text{ and } \ \sigma_1(e_2) = e_{p_2}.$$

The witnesses needed for soundness are σ and $(e_{p_0} \ e_{p_1})$. After using the transitivity of restrict and the **P-App** constructor, the only goal left to prove is that:

$$e_{p_2}[v_{p_0}/f, v_{p_1}/x] \Downarrow v_p.$$

Applying Substitution Interaction twice concludes the proof.

Case N-Bang: We know that

$$e \dashv \kappa \Downarrow_q^t \kappa_a \vDash v \ \text{ and } \ sampleV \ \kappa_a \ v \Rightarrow_{q'}^{t'} \kappa'.$$

By the specification of *sample*, since $\sigma' \in [\![\kappa']\!]$ we know that $\sigma' \in [\![\kappa_a]\!]$ and the result follows directly from the induction hypothesis.

Case N-Narrow: The 4 derived cases from inlining *choose* flow similarly, so without loss of generality let's assume that the first *choose* rule was used. We know

a lot of things from the narrowing derivation:

$$
\begin{gathered}
e \dashv \kappa \Downarrow_q^t \kappa_a \vDash v, \\
e_1 \dashv \kappa_a \Downarrow_{q_1}^{t_1} \kappa_b \vDash v_1, \quad e_2 \dashv \kappa_b \Downarrow_{q_2}^{t_2} \kappa_c \vDash v_2, \\
sampleV\ \kappa_c\ v_1 \Rightarrow_{q_1'}^{t_1'} \kappa_d, \quad sampleV\ \kappa_d\ v_2 \Rightarrow_{q_2'}^{t_2'} \kappa_e, \\
nat_{\kappa_e}(v_1) = n_1, \quad n_1 > 0, \quad nat_{\kappa_e}(v_2) = n_2, \quad n_2 > 0, \\
(\kappa_0, [u_1, u_2]) = fresh\ \kappa_e\ [\overline{T}_1, \overline{T}_2], \\
\kappa_l = unify\ \kappa_0\ v\ (L_{\overline{T}_1 + \overline{T}_2}\ u_1), \\
\kappa_r = unify\ \kappa_0\ v\ (R_{\overline{T}_1 + \overline{T}_2}\ u_2).
\end{gathered}
$$

By the specification of *unify* for the definition of κ_l we know that $\sigma'|_{dom(\kappa_0)} \in [\![\kappa_0]\!]$. By using the specification of *fresh* we can obtain that $\sigma'|_{dom(\kappa_e)} \in [\![\kappa_e]\!]$. Inversion of $nat_{\kappa_e}(v_1) = n_1$ yields that there exists v_{p_1} such that $\kappa_e[v_1] = v_{p_1}$ (where we lift the $\kappa[\cdot]$ notation to values) and similarly for v_{p_2} (using Lemma 4.3.1.7 to preserve the first result). But that means, for all $\sigma \in [\![\kappa_e]\!]$ (including $\sigma'|_{dom(\kappa_e)}$), we have $\sigma(v_i) = v_{p_i}$.

That allows us to use the inductive hypotheses for e_1 and e_2 yielding σ_1, e_{p_1} and e_{p_2} such that $\sigma'|_{\sigma_1} \equiv \sigma_1$, $\sigma_1(e_i) = e_{p_i}$ and $e_{p_i} \Downarrow v_{p_i}$.

Finally, we use the last inductive hypothesis to obtain σ and e_p as appropriate and provide σ and $e_p \leftarrow (e_{p_1}, e_{p_2})$ as witnesses to the entire case. The result follows immediately. □

Theorem A.0.0.9 (Completeness)**.**

$$
\left.\begin{array}{l}
e_p \Downarrow v_p \\
\sigma(e) = e_p \\
\sigma \in [\![\kappa]\!] \wedge \vdash \kappa \\
\varnothing; U(\kappa) \vdash e : T
\end{array}\right\} \Rightarrow
\begin{array}{l}
\exists v\ \kappa'\ \sigma'\ q\ t. \\
\left\{\begin{array}{l}
\sigma'|_\sigma \equiv \sigma \quad \wedge \quad \sigma' \in [\![\kappa']\!] \\
\sigma'(v) = v_p \\
e \dashv \kappa \Downarrow_q^t \kappa' \vDash v
\end{array}\right.
\end{array}
$$

Proof: By induction on the derivation of the predicate semantics judgment.

Case P-Val: The witnesses for completeness are v, κ, σ, 1 and ϵ. The result holds trivially.

Case P-Pair: We have

$$e_{p_1} \Downarrow v_{p_1} \text{ and } e_{p_2} \Downarrow v_{p_2}.$$

By inversion on the substitution we have two cases. In the simple case, e is some unknown u and $\sigma(u) = (e_{p_1}, e_{p_2})$. But then e_{p_1} and e_{p_2} must be values, and therefore $e_{p_i} = v_{p_i}$. By the **N-Base** rule, $u \dashv \kappa \Downarrow_1^{\epsilon} \kappa \vDash u$ and the result follows directly.

In the more interesting case, e is a pair (e_1, e_2) and we know that $\sigma(e_1) = e_{p_1}$ and $\sigma(e_2) = e_{p_2}$. Inverting the typing relation gives us

$$U(\kappa) \vdash e_1 : T_1 \text{ and } U(\kappa) \vdash e_2 : T_2.$$

The inductive hypothesis for the first predicate semantics derivation, instantiated at σ, κ and T_1 gives us that

$$\left.\begin{array}{l} \sigma \in [\![\kappa]\!] \\ \sigma(e_1) = e_{p_1} \\ \vdash \kappa \\ U(\kappa) \vdash e_1 : T_1 \end{array}\right\} \Rightarrow \begin{array}{l} \exists v_1\ \kappa_1\ \sigma_1\ q_1\ t_1. \\ \left\{\begin{array}{l} \sigma_1|_\sigma \equiv \sigma \ \wedge\ \sigma_1 \in [\![\kappa_1]\!] \\ e_1 \dashv \kappa \Downarrow_{q_1}^{t_1} \kappa_1 \vDash v_1 \\ \sigma_1(v_1) = v_{p_1} \end{array}\right. \end{array}.$$

Its assumptions already hold so we can obtain such witnesses. By the second inductive hypothesis, we know that

$$\left.\begin{array}{l} \sigma_1 \in [\![\kappa_1]\!] \\ \sigma_1(e_1) = e_{p_1} \\ \vdash \kappa_1 \\ U(\kappa_1) \vdash e_1 : T_2 \end{array}\right\} \Rightarrow \begin{array}{l} \exists v_2\ \kappa_2\ \sigma_2\ q_2\ t_2. \\ \left\{\begin{array}{l} \sigma_2|_{\sigma_1} \equiv \sigma_1 \ \wedge\ \sigma_2 \in [\![\kappa_2]\!] \\ e_1 \dashv \kappa_1 \Downarrow_{q_2}^{t_2} \kappa_2 \vDash v_1 \\ \sigma_2(v_1) = v_{p_2} \end{array}\right. \end{array}.$$

Since $\sigma_1|_\sigma$ and $\sigma(e_1) = e_{p_1}$, then $\sigma_1(e_1) = e_{p_1}$. By preservation, we get that κ_1 is well typed. Finally, narrowing only extends the typing environment (Lemma A.0.0.1) and then by Unknown Invariance (Lemma A.0.0.2) we can obtain the last assumption $U(\kappa) \vdash e_1 : T_1$ of the inductive hypothesis.

We combine the results from the two inductive hypotheses to provide witnesses

for the existentials:

$$(v_1, v_2),\ \ \kappa_2,\ \ \sigma_2,\ \ q_1 * q_2 \text{ and } t_1 \cdot t_2.$$

By transitivity of restrict, we get that $\sigma \equiv \sigma_2|_\sigma$, while the inclusion property $\sigma_2 \in [\![\kappa_2]\!]$ is satisfied by the inductive hypothesis directly. To prove that $\sigma_2((v_1, v_2)) = (v_{p_1}, v_{p_2})$ we just need to prove that $\sigma_2(v_1) = v_{p_1}$, but that holds because $\sigma(e_1) = e_{p_1}$ and σ is a restriction of σ_2. Using the **N-Pair** constructor completes the proof.

Case P-App: We know that

$$e_{p_0} \Downarrow v_{p_0},\ e_{p_1} \Downarrow v_{p_1} \text{ and } e_{p_2}[v_{p_0}/f, v_{p_1}/x] \Downarrow v_p,$$

for some $v_{p_0} = (rec\ (f : T_1 \to T_2)\ x = e_{p_2})$. Inversion on the substitution gives us only one possible e, since unknowns only range over values: $e = (e_0\ e_1)$, where $\sigma(e_0) = e_{p_0}$ and $\sigma(e_1) = e_{p_1}$. Inversion of the typing premise gives us

$$U(\kappa) \vdash e_0 : T_1' \to T_2' \text{ and } U(\kappa) \vdash e_1 : T_1'.$$

By the inductive hypothesis for the derivation of e_{p_0} we get that

$$\left.\begin{array}{l} \sigma \in [\![\kappa]\!] \\ \sigma(e_0) = e_{p_0} \\ \vdash \kappa \\ U(\kappa) \vdash e_0 : T_1' \to T_2' \end{array}\right\} \Rightarrow \begin{array}{l} \exists v_0\ \kappa_0\ \sigma_0\ q_0\ t_0. \\ \left\{\begin{array}{l} \sigma_0|_\sigma \equiv \sigma \ \ \wedge \ \ \sigma_0 \in [\![\kappa_0]\!] \\ e_0 \rightrightarrows \kappa \Downarrow_{q_0}^{t_0} \kappa_0 \vDash v_0 \\ \sigma_0(v_0) = v_{p_0} \end{array}\right. \end{array}.$$

All its assumptions hold, so we can invert the last substitution to obtain that $v_0 = (rec\ (f : T_1 \to T_2)\ x = e_2)$, where $\sigma_0(e_2) = e_{p_2}$. By preservation, we know that the type of v_0 is the type of e_0 in κ_0 and uniqueness of typing equates T_1 with T_1' and T_2 with T_2'.

By the second inductive hypothesis, we know that

$$\left.\begin{array}{l}\sigma_0 \in [\![\kappa_0]\!] \\ \sigma_0(e_1) = e_{p_1} \\ \vdash \kappa_0 \\ U(\kappa_0) \vdash e_1 : T_1\end{array}\right\} \Rightarrow \begin{array}{l}\exists v_1\ \kappa_1\ \sigma_1\ q_1\ t_1. \\ \left\{\begin{array}{l}\sigma_1|_{\sigma_0} \equiv \sigma_0 \ \ \wedge \ \ \sigma_1 \in [\![\kappa_1]\!] \\ e_1 \dashv \kappa_0 \Downarrow_{q_1}^{t_1} \kappa_1 \vDash v_1 \\ \sigma_1(v_1) = v_{p_1}\end{array}\right.\end{array}.$$

As in **P-Pair** we can discharge all of its assumptions.

Let $e_2' = e_2[v_0/f, v_1/x]$ and $e_{p_2'} = e_{p_2}[v_{p_0}/f, v_{p_1}/x]$. The last inductive hypothesis states that

$$\left.\begin{array}{l}\sigma_1 \in [\![\kappa_1]\!] \\ \sigma_1(e_2') = e_{p_2'} \\ \vdash \kappa_1 \\ U(\kappa_1) \vdash e_2' : T_2\end{array}\right\} \Rightarrow \begin{array}{l}\exists v_2'\ \kappa_2'\ \sigma_2'\ q_2'\ t_2'. \\ \left\{\begin{array}{l}\sigma_2'|_{\sigma_1} \equiv \sigma_1 \ \ \wedge \ \ \sigma_2' \in [\![\kappa_2']\!] \\ e_2' \dashv \kappa_1 \Downarrow_{q_2'}^{t_2'} \kappa_2' \vDash v_2' \\ \sigma_2'(v_2') = v\end{array}\right.\end{array}.$$

The substitution premise can be discharged using the Substitution Interaction Lemma (Lemma A.0.0.4), while the typing premise by repeated applications of the Substitution Lemma and Unknown Invariance. The proof concludes by combining the probabilities and traces by multiplication and concatenation respectively.

Cases P-L, P-R, P-Fold, P-After: These cases are similar to **P-Pair**.

Case P-CasePair: From the predicate derivations we have that

$$e_p \Downarrow (v_{p_1}, v_{p_2}) \text{ and } e_p'[v_{p_1}/x, v_{p_2}/y] \Downarrow v_p'.$$

Inverting the substitution premise leaves us with $\sigma(e) = e_p$ and $\sigma(e') = e_p'$, while inverting the typing premise yields

$$U(\kappa) \vdash e : T_1 + T_2 \text{ and } (x \mapsto T_1, y \mapsto T_2); U(\kappa) \vdash e' : T.$$

The inductive hypothesis for e_p gives us that

$$\left.\begin{array}{l}\sigma \in [\![\kappa]\!] \\ \sigma(e) = e_p \\ \vdash \kappa \\ U(\kappa) \vdash e : T_1 + T_2\end{array}\right\} \Rightarrow \begin{array}{l}\exists v_1\ \kappa_1\ \sigma_1\ q_1\ t_1. \\ \left\{\begin{array}{l}\sigma_1|_\sigma \equiv \sigma \quad \wedge \quad \sigma_1 \in [\![\kappa_1]\!] \\ e \dashv \kappa \Downarrow_{q_1}^{t_1} \kappa_1 \vDash v \\ \sigma_1(v) = (v_{p_1}, v_{p_2})\end{array}\right.\end{array}.$$

To decide which of **N-CasePair-P** and **N-CasePair-U** we will use, we invert the substitution relation for v. In the simple case, $v = (v_1, v_2)$ and the proof flows similarly to **P-App**.

The interesting case is when v is an unknown u, in which case we need to "build up" the derivation of **N-CasePair-U**. Let

$$\begin{gathered}(\kappa_{1a}, [u_1, u_2]) = \mathit{fresh}\ \kappa_1\ [T_1, T_2], \\ \kappa_{1b} = \mathit{unify}\ \kappa_{1a}\ (u_1, u_2)\ u, \\ \sigma_1' = \sigma_1 \oplus u_1 \mapsto v_1 \oplus u_2 \mapsto v_2, \\ e'' = e'[u_1/x, u_2/y] \text{ and } e_p'' = e_p'[v_{p_1}/x, v_{p_2}/y].\end{gathered}$$

The second inductive hypothesis (instantiated at σ_1', κ_{1b}) states that

$$\left.\begin{array}{l}\sigma_1' \in [\![\kappa_{1b}]\!] \\ \vdash \kappa_{1b} \\ U(\kappa_{1b}) \vdash e'' : T \\ \sigma_1'(e'') = e_p''\end{array}\right\} \Rightarrow \begin{array}{l}\exists v'\ \kappa_2\ \sigma_2\ q_2\ t_2. \\ \left\{\begin{array}{l}\sigma_2|_{\sigma_1'} \equiv \sigma_1' \quad \wedge \quad \sigma_2 \in [\![\kappa_2]\!] \\ \sigma(v') = v_p' \\ e'' \dashv \kappa_{1b} \Downarrow_{q_2}^{t_2} \kappa_2 \vDash v'\end{array}\right.\end{array}.$$

To use this inductive hypothesis we need to discharge all of its assumptions first.

To prove that $\sigma_1' \in [\![\kappa_{1b}]\!]$ we start with $\sigma_1 \in [\![\kappa_1]\!]$ by the first induction hypothesis. By the specification of *fresh*, the denotation of κ_1 remains unchanged, therefore $\sigma_1 \in [\![\kappa_{1a}]\!]$. Since u_1, u_2 are not in the domain of κ_{1a}, the restriction $\sigma_1'|_{dom(\kappa_{1a})}$ is σ_1. Therefore, by the specification of *unify* we just need to show that

$\sigma_1'(u) = \sigma_1'((u_1, u_2))$. Indeed,

$$\begin{aligned}\sigma_1'(u) = \sigma_1(u) = (v_{p_1}, v_{p_2}) &= (\sigma_1'(u_1), \sigma_1'(u_2)) \\ &= \sigma_1'((u_1, u_2)),\end{aligned}$$

which concludes the proof of the first premise.

The fact that κ_{1b} is well typed is a direct corollary of the typing lemmas for *fresh* (Lemma 4.3.1.4) and *unify* (Lemma 4.3.1.9).

To prove that $U(\kappa_{1b}) \vdash e'' : T$ we apply the Substitution Lemma twice. Then we need to prove that

$$\begin{gathered}\varnothing; U(\kappa_{1b}) \vdash u_i : T_i \text{ for } i = 1, 2 \\ \text{and } (x \mapsto T_1, y \mapsto T_2); U(\kappa_{1b}) \vdash e' : T.\end{gathered}$$

By the specification of *unify* we know that $U(\kappa_{1b}) = U(\kappa_{1a})$, while from the specification of *fresh* we obtain

$$U(\kappa_{1a}) = U(\kappa_1) \oplus u_1 \mapsto T_1 \oplus u_2 \mapsto T_2.$$

This directly proves the former results for u_1, u_2, while Unknown Invariance (since $U(\kappa_{1b})$ is an extension of $U(\kappa)$) proves the latter.

The final premise of the inductive hypothesis, $\sigma_1'(e'') = e_p''$, is easily proved by applying the Substitution Interaction lemma (Lemma A.0.0.4) twice.

Since we have satisfied all of its premises, we can now use the result of the second inductive hypothesis. It provides most of the witnesses to completeness (v, κ_2 and σ_2), while, as usual, we combine the probabilities and traces by multiplying and concatenating them. The result follows by transitivity of restrict and use of the **N-CasePair-U** constructor.

Cases P-Case-L, P-Case-R, P-Unfold: These cases are in direct correspondence with **P-CasePair**. The only difference is that to choose between which *choose* rule to follow we case analyze on the satisfiability of the corresponding constraint set.

Case P-Bang: We know that $e_p \Downarrow v_p$. By the inductive hypothesis, we immediately obtain that there exists some v, σ_1, κ_1, q_1 and t_1 such that $\sigma_1(v) = v_p$ and $\sigma_1 \in [\![\kappa]\!]$ and $\sigma_1|_\sigma \equiv \sigma$ and that $e \dashv \kappa \Downarrow_{q_1}^{t_1} \kappa_1 \vDash v$. By the completeness requirement of *sample* lifted to *sample*V, we know that there exists some q_2, t_2 and κ_2 such that *sample*V κ_1 $v \Rightarrow_{q_2}^{t_2} \kappa_2$ and $\sigma_1 \in [\![\kappa_2]\!]$. The result follows easily.

Case P-Narrow: We know that

$$e_p \Downarrow v_p, \;\; e_{p_1} \Downarrow v_{p_1} \text{ and } e_{p_2} \Downarrow v_{p_2},$$

while $[\![v_{p_1}]\!] > 0$ and $[\![v_{p_2}]\!] > 0$. Chaining the induction hypothesis as in **P-Pair**, we get that there exist $v, v_1, v_2, \sigma', \kappa_1, \kappa_2, \kappa', q, q_1, q_2, t, t_1$ and t_2 such that

$$\sigma' \in [\![\kappa']\!], \quad \begin{array}{cc} \sigma'|_\sigma \equiv \sigma & e \dashv \kappa \Downarrow_q^t \kappa_1 \vDash v \\ \sigma'(v) = v_p \text{ and} & e_1 \dashv \kappa_1 \Downarrow_{q_1}^{t_1} \kappa_2 \vDash v_1 \\ \sigma'(v_i) = v_{p_i} & e_2 \dashv \kappa_2 \Downarrow_{q_2}^{t_2} \kappa_3 \vDash v_2 \end{array}$$

By the lifted completeness requirement of *sample*, we know that there exist $q_1', q_2', t_1', t_2', \kappa_4$ and κ_5 such that $\sigma \in [\![\kappa_5]\!]$,

$$\mathit{sample}V \; \kappa_3 \; v_1 \Rightarrow_{q_1'}^{t_1'} \kappa_4 \text{ and } \mathit{sample}V \; \kappa_4 \; v_2 \Rightarrow_{q_2'}^{t_2'} \kappa_5 \; .$$

By definition, $\mathit{nat}_{\kappa_5}(v_1) = v_{p_1}$ and $\mathit{nat}_{\kappa_5}(v_2) = v_{p_2}$.

Without loss of generality, assume that $v_p = L \; v_p'$ for some v_p' and let

$$\sigma'' = \sigma' \oplus u_1 \mapsto v_p' \text{ and } (\kappa', [u_1, u_2]) = \mathit{fresh} \; \kappa_5 \; [T_1, T_2]$$

and

$$\kappa_l = \mathit{unify} \; \kappa' \; (L \; u_1) \; v \text{ and } \kappa_r = \mathit{unify} \; \kappa' \; (R \; u_2) \; v.$$

By transitivity of restrict, $\sigma''|_\sigma \equiv \sigma$. Moreover, $\sigma''(v) = v_p$. The proof that $\sigma'' \in [\![\kappa_l]\!]$ is similar to the proof that $\sigma_1' \in [\![\kappa_{1b}]\!]$ in **P-Pair**. To conclude the proof, we case analyze on whether κ_r is satisfiable or not and choosing which *choose* derivation to follow accordingly. □

Preservation is simpler than before since we only deal with a single output. We still need a similar lemma about the effect of the matching semantics on types:

Lemma A.0.0.10 (Matching Effect on Types)**.**

$$p \Leftarrow e \dashv \kappa \Uparrow_q^t v\{\kappa'\} \;\Rightarrow\; U(\kappa')|_{U(\kappa)} \equiv U(\kappa)$$

Proof: By induction on the derivation and transitivity of restrict. □

Theorem A.0.0.11 (Preservation)**.**

$$\left.\begin{array}{l} p \Leftarrow e \dashv \kappa \Uparrow_q^t \{\kappa'\} \\ U(\kappa) \vdash e : \overline{T} \\ U(\kappa) \vdash p : \overline{T} \\ \vdash \kappa \end{array}\right\} \Rightarrow \vdash \kappa'$$

Proof:

Case M-Base: Follows directly from the typing lemma of *unify* (Lemma 4.3.1.9).

Case M-Pair: We know that

$$u_1 \Leftarrow e_1 \dashv \kappa_0 \Uparrow_{q_1}^{t_1} \{\kappa_1\} \text{ and } u_2 \Leftarrow e_2 \dashv \kappa_1 \Uparrow_{q_2}^{t_2} \{\kappa_2\},$$

where

$$(\kappa', [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2],$$
$$\kappa_0 = \mathit{unify}\ \kappa'\ (u_1, u_2)\ p.$$

By inversion of the typing relation for (e_1, e_2) we know that

$$U(\kappa) \vdash e_1 : \overline{T}_1 \text{ and } U(\kappa) \vdash e_2 : \overline{T}_2.$$

Based on the specification of *fresh*,

$$U(\kappa') = U(\kappa) \oplus u_1 \mapsto \overline{T}_1 \oplus u_2 \mapsto \overline{T}_2 \text{ and } \vdash \kappa',$$

while *unify* preserves all type information. Therefore, κ_0 is well typed and $U(\kappa_0) \vdash u_1 : \overline{T}_1$ and $U(\kappa_0) \vdash u_2 : \overline{T}_2$. By Unknown Invariance (Lemma A.0.0.2) e_1 and e_2 are well typed in κ_0 as well.

Now we can use the inductive hypothesis for the derivation of u_1 which gives us that $\vdash \kappa_1$. To conclude the proof, we can use the other inductive hypothesis; for that we just need to show that $U(\kappa_1) \vdash u_2 : \overline{T}_2$ and $U(\kappa_1) \vdash e_2 : \overline{T}_2$. However, by the typing lemma for the matching semantics (Lemma A.0.0.10) we known that $U(\kappa_1)|_{U(\kappa_0)} \equiv U(\kappa_0)$. Unknown Invariance completes this case.

Case M-CasePair: We know that

$$(u_1, u_2) \Leftarrow e \rightleftharpoons \kappa_a \Uparrow_{q_1}^{t_1} \{\kappa_b\} \text{ and } p \Leftarrow e'' \rightleftharpoons \kappa_b \Uparrow_{q_2}^{t_2} \{\kappa'\},$$

where

$$e'' = e'[u_1/x, u_2/y],$$
$$(\kappa_a, [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2].$$

Like in the **M-Pair** case, using the definition of *fresh* we can obtain that $U(\kappa') = U(\kappa) \oplus u_1 \mapsto \overline{T}_1 \oplus u_2 \mapsto \overline{T}_2$ as well as $\vdash \kappa_a$ and therefore $U(\kappa_a) \vdash (u_1, u_2) : \overline{T}_1 \times \overline{T}T_2$. We again can invert the typing relation for the entire case to obtain that

$$U(\kappa) \vdash e : \overline{T}'_1 \times \overline{T}'_2 \text{ and } x \mapsto \overline{T}'_1, y \mapsto \overline{T}'_2; U(\kappa) \vdash e' : \overline{T},$$

while type uniqueness equates $\overline{T}_i$ with $\overline{T}'_i$. Using Unknown Invariance we can propagate these typing relations to κ_a.

We can now use the inductive hypothesis on the matching derivation for e to obtain that κ_b is well typed. By the typing lemma for the matching semantics and Unknown Invariance we lift all typing relations to κ_b. To conclude the proof using the second inductive hypothesis we need only prove that $U(\kappa_b) \vdash e'[u_1/x, u_2/y] : \overline{T}$, which follows by consecutive applications of the Substitution Lemma.

Cases M-L-Sat, M-R-Sat, M-Fold: Follow similarly to **M-Pair**.

Case M-App: For some $v_0 = (rec\ f\ x = e_2)$, we have that

$$e_0 \rightleftharpoons \kappa \Downarrow_{q_0}^{t_0} \kappa_0 \vDash v_0 \text{ and } e_1 \rightleftharpoons \kappa_0 \Downarrow_{q_1}^{t_1} \kappa_1 \vDash v_1,$$

while

$$p \Leftarrow e'[v_0/f, v_1/x] \rightleftharpoons \kappa_1 \Uparrow_{q_2}^{t_2} \{\kappa'\}.$$

By inverting the typing relation for $e_0\ e_1$ we get that $U(\kappa) \vdash e_0 : T \to \overline{T}$ and $U(\kappa) \vdash e_1 : T$. Using the preservation theorem for the narrowing semantics (Theorem A.0.0.3) we know that κ_0 is well typed and the lambda has the same type as e_0 in κ. That means that

$$(f \mapsto (T \to \overline{T}), x \mapsto T); U(\kappa_0) \vdash e_2 : \overline{T}.$$

The typing lemma for the narrowing semantics (Lemma A.0.0.1) and Unknown Invariance allow us to lift type information to κ_0. We repeat this process for the second narrowing derivation. To use the inductive hypothesis and conclude the proof, we only need to apply the Substitution Lemma twice as in **M-CasePair**.

Case M-Unfold: This case follows directly from the induction hypothesis.

Case M-After: We know that

$$p \Leftarrow e_1 \rightleftharpoons \kappa \Uparrow_{q_1}^{t_1} \{\kappa_1\} \text{ and } e_2 \rightleftharpoons \kappa_1 \Downarrow_{q_2}^{t_2} \kappa_2 \vDash v.$$

As in **M-Pair**, we invert the typing relation to obtain type information for e_1 and e_2. We the use the inductive hypothesis on the first derivation to obtain that κ_1 is well typed. To conclude the proof, we need only apply the preservation lemma for the narrowing semantics, and its premise that e_2 is well typed is discharged as usual using the typing lemma for the matching semantics and Unknown Invariance.

Cases M-Pair-Fail, M-CasePair-Fail, M-After-Fail, M-L-UnSat, M-R-UnSat, M-Case-4, M-Bang-Fail, M-Narrow-Fail: These cases are vacuously true since no constraint set is returned.

Cases M-CasePair-Fun, M-Case-L-Fun, M-Case-R-Fun: Similar to **M-App**.

Case M-Bang: We know that

$$p \Leftarrow e \dashv \kappa \Uparrow_{q_1}^{t_1} \{\kappa_1\},$$

where

$$sampleV\ \kappa_1\ p \Rightarrow_{q_2}^{t_2} \kappa'.$$

By the inductive hypothesis we immediately get that κ_1 is well typed. The specification of *sample* lifted to *sampleV* yields the result.

Case M-Narrow: We know that

$$p \Leftarrow e \dashv \kappa \Uparrow_{q}^{t} \{\kappa_a\}$$

and

$$e_1 \dashv \kappa_a \Downarrow_{q_1}^{t_1} \kappa_b \vDash v_1 \text{ and } e_2 \dashv \kappa_b \Downarrow_{q_2}^{t_2} \kappa_c \vDash v_2.$$

As in the previous cases, we use the inductive hypothesis and the preservation lemma for the narrowing semantics to ensure all variables are appropriately typed in κ_c.

Following the matching judgment, we proceed to *sampleV* twice resulting in a constraint set κ_e; as in **M-Bang**, κ_e is well typed. We then generate two unknowns u_1 and u_2 with types $\overline{T}_1$ and $\overline{T}_2$ to obtain a constraint set κ_0, that is well typed because of the specification of *fresh*. Finally, we *unify* the pattern p with the fresh unknowns tagged L or R, yielding κ_l and κ_r that are both well typed because of the specification of *unify*. Since all *choose* does is pick which of κ_l and κ_r to return, the result follows immediately.

Cases M-Case-1, M-Case-2, M-Case-3: These cases flow similarly, using repeated applications of the inductive hypotheses. The only case that hasn't been encountered in a previous rule is for **M-Case-1**, when both branch derivations yield some (well-typed) constraint sets $\{\kappa_a\}$ and $\{\kappa_b\}$ that are combined using *union*. But by the typing lemma for *union*, its result is also well typed. □

Theorem A.0.0.12 (Soundness)**.**

$$\left.\begin{array}{l} p \Leftarrow e \dashv \kappa \Uparrow_q^t \{\kappa'\} \\ \sigma'(p) = v_p \wedge \sigma' \in [\![\kappa']\!] \\ \forall u.\ (u \in e \vee u \in p) \Rightarrow u \in dom(\kappa) \end{array}\right\} \Rightarrow \exists \sigma\ e_p. \left\{\begin{array}{l} \sigma'|_\sigma \equiv \sigma \\ \sigma \in [\![\kappa]\!] \\ \sigma(e) = e_p \\ e_p \Downarrow v_p \end{array}\right.$$

Proof: By induction on the matching derivation, following very closely the structure of proof of soundness for the narrowing semantics: we use the inductive hypothesis for every matching derivation in reverse order, obtaining witnesses for valuations and expressions, while concluding the proof with the specifications of constraint set operations and transitivity.

Case M-Base: In the base case, just like in the proof for the **N-Base** rule, the witnesses are σ' and v_p. The inclusion $\sigma' \in \kappa$ is a direct result of the specification of *unify*.

Case M-Pair: We know that

$$u_1 \Leftarrow e_1 \dashv \kappa_0 \Uparrow_{q_1}^{t_1} \{\kappa_1\} \text{ and } u_2 \Leftarrow e_2 \dashv \kappa_1 \Uparrow_{q_2}^{t_2} \{\kappa'\},$$

where

$$(\kappa_a, [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2]$$

and

$$\kappa_0 = \mathit{unify}\ \kappa_a\ (u_1, u_2)\ p.$$

By the definition of *fresh* and the fact that the domain is increasing, we know that u_2 is in the domain of κ'. That means that there exists some value v'_{p_2} such that $\sigma'(u_2) = v'_{p_2}$. By the inductive hypothesis for σ' and u_2 we get that there exist some σ_1 and e_{p_2} such that σ_1 is a restriction of σ' in κ_1, while

$$\sigma_1(e_2) = e_{p_2} \text{ and } e_{p_2} \Downarrow v'_{p_2}.$$

Using a similar argument to obtain a v'_{p_1} such that $\sigma_1(u_1) = v'_{p_1}$, we can leverage the inductive hypothesis again on the first derivation gives us that there exists some

σ and e_{p_1} such that σ is a restriction of σ_1 in κ_0 and

$$\sigma(e_1) = e_{p_1} \text{ and } e_{p_1} \Downarrow v'_{p_1}.$$

Our soundness witnesses are σ and (e_{p_1}, e_{p_2}). By the specification of *unify* we know that $\sigma(p) = \sigma((u_1, u_2))$ and decreasingness helps us conclude that $v_p = (v'_{p_1}, v'_{p_2})$ which concludes the proof of the pair case, along with transitivity of valuation restriction.

Cases M-Case-1, M-Case-2, M-Case-3: The only new rules are the case rules; however, the general structure of the proof is once again similar. For **M-Case-1**, we know that:

$$(\kappa_0, [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2],$$

$$(L_{\overline{T}_1+\overline{T}_2}\ u_1) \Leftarrow e \dashv \kappa_0 \Uparrow_{q_1}^{t_1} \{\kappa_1\},$$

$$(R_{\overline{T}_1+\overline{T}_2}\ u_2) \Leftarrow e \dashv \kappa_0 \Uparrow_{q_2}^{t_2} \{\kappa_2\},$$

$$p \Leftarrow e_1[u_1/x_l] \dashv \kappa_1 \Uparrow_{q'_1}^{t'_1} \kappa_a^? \text{ and } p \Leftarrow e_2[u_2/y_r] \dashv \kappa_2 \Uparrow_{q'_2}^{t'_2} \kappa_b^?,$$

while

$$\kappa^? = \mathit{combine}\ \kappa_0\ \kappa_a^?\ \kappa_b^?.$$

If either of the non-union *combine* cases fire, the proof is simple. If $\kappa_a^? = \kappa_b^? = \varnothing$, then there exists no κ' such that the result of the derivation is $\{\kappa'\}$.

Let's assume that $\kappa_a^? = \{\kappa_a\}$ for some κ_a and $\kappa_b^? = \varnothing$ (the symmetric case follows similarly). Then we know that $\sigma' \in [\![\kappa_a]\!]$ and from the inductive hypothesis for the e_1 derivation we get that there exist σ_1 and e_{p_1} such that $\sigma \in [\![\kappa_1]\!]$ and $\sigma_1(e_1[u_1/x_l]) = e_{p_1}$. As in the narrowing soundness proof, we can leverage the inverse substitution interaction lemma (A.0.0.4) to conclude that there exists some e'_1 such that $\sigma_1(e_1) = e'_1$. An additional application of the inductive hypothesis for the evaluation of e against the $L_{\overline{T}_1+\overline{T}_2}\ u_1$ gives us σ and e_p such that $\sigma \in [\![\kappa]\!]$ and $\sigma(e) = e_p$, which are also the soundness witnesses that conclude the proof.

The more interesting case is when $\kappa_a^? = \{\kappa_a\}$ and $\kappa_b^? = \{\kappa_b\}$ for some constraint sets κ_a and κ_b. In that case, $\sigma' \in [\![\kappa_a]\!]$ or $\sigma' \in [\![\mathit{rename}\ (U(\kappa_a)\text{-}U(\kappa_0))\ \kappa_b]\!]$. The first

case proceeds exactly like the one for $\kappa_b^? = \varnothing$. For the latter, we need to push the renaming to σ', obtaining some σ_r which is an alpha-converted version of σ' and then proceed similarly. Since the alpha conversion only happens in the unknowns that are not present in the original constraint set, the choice of these unknowns doesn't matter for the final witness. □

Before we go to completeness we need an auxiliary lemma that ensures there exists *some* derivation that returns a constraint set option if this requirement holds. This is only necessary for the combining **M-Case-1**.

Lemma A.0.0.13 (Termination)**.**

$$\left.\begin{array}{l} \varnothing; U(\kappa) \vdash e : \overline{T} \ \wedge \ \vdash \kappa \\ \forall \sigma \in [\![\kappa]\!].\ \exists v'.\ \sigma(e) \Downarrow v' \end{array}\right\} \Rightarrow \exists \kappa^?\ q\ t.\ p \Leftarrow e \dashv \kappa \Uparrow_q^t \kappa^?$$

The proof of this lemma is almost identical to the completeness proof. Since it doesn't require or enforce particular valuation memberships of the constraint sets involved, every case can follow with the same argument. The only rules where the difference matters is in the *case* rules, where the lack of assumptions allows to provide some termination witness without guaranteeing that the resulting constraint set is not $\varnothing$.

We also need another straightforward lemma regarding the completeness of values:

Lemma A.0.0.14 (Value Completeness)**.**

$$\left.\begin{array}{l} U(\kappa) \vdash e : \overline{T} \\ \vdash \kappa \\ \sigma \in [\![\kappa]\!] \\ \sigma(e) = v_p \\ \sigma(p) = v_p \end{array}\right\} \Rightarrow \exists \kappa'\ \sigma'\ q\ t. \left\{\begin{array}{l} \sigma'|_\sigma \equiv \sigma \\ \sigma' \in [\![\kappa']\!] \\ p \Leftarrow e \dashv \kappa \Uparrow_q^t \{\kappa'\} \end{array}\right.$$

Proof: By induction on e.

Case $e = ()$ ***or*** $e = u$***:*** If e was unit or an unknown, let $\kappa' = \mathit{unify}\ \kappa\ e\ p$. By

the specification of *unify* $\sigma \in [\![\kappa']\!]$. The witnesses to conclude the case are $\kappa',\ \sigma,\ 1$ and ϵ using the **M-Base** rule.

Case $e = (e_1, e_2)$***:*** Following the **M-Pair** rule, let

$$(\kappa', [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2]$$

and

$$\kappa_0 = \mathit{unify}\ \kappa'\ (u_1, u_2)\ p.$$

We invert the substitution relation to obtain that $\sigma(e_1) = v_{p_1}$ and $\sigma(e_2) = v_{p_2}$ for some v_{p_1}, v_{p_2}. Let $\sigma' = \sigma \oplus u_1 \mapsto v_{p_1} \oplus u_2 \mapsto v_{p_2}$. Then since u_1 and u_2 are fresh, $\sigma'|_\sigma \equiv \sigma$ and, by the specification of *unify*, $\sigma' \in \kappa'$.

By the inductive hypothesis for e_1 (inverting the typing relation for the typing premise), there exist $\sigma_1,\ \kappa_1,\ q_1$ and t_1 such that $\sigma_1|_{\sigma'} \equiv \sigma'$ and $\sigma_1 \in [\![\kappa_1]\!]$ and

$$u_1 \Leftarrow e_1 \dashv \kappa_0 \Uparrow_{q_1}^{t_1} \kappa_1.$$

Using Unknown Invariance we can apply the second inductive hypothesis to get similar $\sigma_2,\ \kappa_2, q_2$ and t_2. We conclude the case by providing the witnesses σ_2 and κ_2, while combining the probabilities and traces as usual ($q_1 * q_2$ and $t_1 \cdot t_2$).

Cases L***,*** R ***or*** *fold****:*** The remaining cases are similar to the pair case, with only one inductive hypothesis.

□

Finally, we will need to propagate the termination information across matching derivations. For that we can prove the following corollary of decreasingness:

Corollary A.0.0.15. Termination Preservation

$$\left.\begin{array}{l} p \Leftarrow e \dashv \kappa \Uparrow_q^t \kappa' \\ \forall \sigma \in [\![\kappa]\!].\ \exists v.\ \sigma(e) \Downarrow v \end{array}\right\} \Rightarrow\ \forall \sigma' \in [\![\kappa']\!].\ \exists v.\ \sigma'(e) \Downarrow v$$

Proof: By decreasingness, $\kappa' \leqslant \kappa$, which means that $\sigma'|_\sigma \in [\![\kappa]\!]$. Then, there exists v such that $\sigma'|_\sigma(e) \Downarrow v$ and the result follows. □

Theorem A.0.0.16 (Completeness)**.**

$$\left.\begin{array}{l} e_p \Downarrow v_p \ \wedge \ \sigma \in [\![\kappa]\!] \\ \varnothing; U(\kappa) \vdash e : \overline{T} \ \wedge \ \vdash \kappa \\ \sigma(e) = e_p \wedge \sigma(p) = v_p \\ \forall \sigma' \in [\![\kappa]\!].\ \exists v'.\ \sigma'(e) \Downarrow v' \end{array}\right\} \Rightarrow \quad \exists \kappa'\ \sigma'\ q\ t. \left\{\begin{array}{l} \sigma'|_\sigma \equiv \sigma \\ \sigma' \in [\![\kappa']\!] \\ p \Leftarrow e \dashv \kappa \Uparrow_q^t \{\kappa'\} \end{array}\right.$$

Proof: By induction on the predicate derivation.

Case P-Val: Follows directly from the completeness lemma for values.

Case P-Pair: We have

$$e_{p_1} \Downarrow v_{p_1} \ \text{ and } \ e_{p_2} \Downarrow v_{p_2}.$$

As in the narrowing proof, we invert the substitution of e and get two cases. In the simple case, e is some unknown u and $\sigma(u) = (e_{p_1}, e_{p_2})$. But then e_{p_1} and e_{p_2} must be values, and the proof follows by the value completeness lemma.

In the more interesting case, e is a pair (e_1, e_2) and we know that $\sigma(e_1) = e_{p_1}$ and $\sigma(e_2) = e_{p_2}$. Inverting the typing relation gives us

$$U(\kappa) \vdash e_1 : \overline{T}_1 \ \text{ and } \ U(\kappa) \vdash e_2 : \overline{T}_2.$$

Following the **M-Pair** rule, let

$$(\kappa', [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2]$$

and

$$\kappa_0 = \mathit{unify}\ \kappa'\ (u_1, u_2)\ p.$$

As in the value completeness lemma, let $\sigma_0 = \sigma \oplus u_1 \mapsto v_{p_1} \oplus u_2 \mapsto v_{p_2}$. By the

inductive hypothesis for the derivation of e_{p_1},

$$\left.\begin{array}{l} U(\kappa_0) \vdash e_1 : \overline{T}_1 \\ \vdash \kappa_0 \ \ \wedge \ \ \sigma_0 \in [\![\kappa_0]\!] \\ \sigma_0(e_1) = e_{p_1} \\ \sigma_0(u_1) = v_{p_1} \\ \forall \sigma' \in [\![\kappa_0]\!].\ \exists v'.\ \sigma'(e_1) \Downarrow v' \end{array}\right\} \Rightarrow \left\{\begin{array}{l} \exists \kappa_1\ \sigma_1\ q_1\ t_1. \\ \left\{\begin{array}{l} \sigma_1|_{\sigma_0} \equiv \sigma_0 \\ \sigma_1 \in [\![\kappa_1]\!] \\ u_1 \Leftarrow e_1 \dashv \kappa_0 \Uparrow_{q_1}^{t_1} \{\kappa_1\} \end{array}\right. \end{array}\right.$$

Since u_1 and u_2 are fresh, we get that $\sigma_0|_\sigma \equiv \sigma$ as well as $\sigma_0((u_1, u_2)) = \sigma_0(p)$. But then by the specification of unify $\sigma_0 \in [\![\kappa_0]\!]$. Moreover, by the ordering lemmas for *fresh* and *unify* (Lemma 4.3.1.2 and Lemma 4.3.1.8) we know that $\kappa_0 \leqslant \kappa$ which means that the termination assumption for valuations in κ is preserved and we can now use the above inductive hypothesis.

The inductive hypothesis for the derivation of e_{p_2} yields

$$\left.\begin{array}{l} U(\kappa_1) \vdash e_2 : \overline{T}_2 \\ \vdash \kappa_1 \ \ \wedge \ \ \sigma_1 \in [\![\kappa_1]\!] \\ \sigma_1(e_2) = e_{p_2} \\ \sigma_1(u_2) = v_{p_2} \\ \forall \sigma' \in [\![\kappa_1]\!].\ \exists v'.\ \sigma'(e_2) \Downarrow v' \end{array}\right\} \Rightarrow \left\{\begin{array}{l} \exists \kappa_2\ \sigma_2\ q_2\ t_2. \\ \left\{\begin{array}{l} \sigma_2|_{\sigma_1} \equiv \sigma_1 \\ \sigma_1 \in [\![\kappa_2]\!] \\ u_2 \Leftarrow e_2 \dashv \kappa_1 \Uparrow_{q_2}^{t_2} \{\kappa_2\} \end{array}\right. \end{array}\right.$$

Like in the narrowing proof, we can discharge the typing hypothesis by using a lemma similar to Lemma A.0.0.1 (which in turn is once again a simple induction on the matching derivation) and Unknown Invariance, while the termination assumption can be discharged using the Termination Preservation corollary.

Our final witnesses are $\sigma_2,\ \kappa_2$ and the standard combinations of probabilities and traces.

Case P-App: We know that

$$e_{p_0} \Downarrow v_{p_0},\ \ e_{p_1} \Downarrow v_{p_1} \text{ and } e_{p_2} \Downarrow v_p,$$

where v_{p_0} is of the form $(rec\ (f : T_1 \to T_2)\ x = e_{p_2})$ and $e'_{p_2} = e_{p_2}[v_{p_0}/f, v_{p_1}/x]$. Inversion on the substitution gives us only one possible

e, since unknowns only range over values: $e = (e_0\ e_1)$, where $\sigma(e_0) = e_{p_0}$ and $\sigma(e_1) = e_{p_1}$. Inversion of the typing premise gives us

$$U(\kappa) \vdash e_0 : T_1' \to \overline{T}_2' \text{ and } U(\kappa) \vdash e_1 : \overline{T}_1'.$$

Using preservation and type uniqueness we can equate T_1 with T_1' as well as T_2 with $\overline{T}_2'$.

We can then turn to the completeness theorem for the narrowing semantics twice to obtain witnesses such that:

$$e_0 \dashv \kappa \Downarrow_{q_0}^{t_0} \kappa_0 \vDash (\mathit{rec}\ (f : T_1 \to \overline{T}_2)\ x = e_2)$$

and

$$e_1 \dashv \kappa_0 \Downarrow_{q_1}^{t_1} \kappa' \vDash v_1.$$

Completeness also guarantees that there exists $\sigma' \in [\![\kappa']\!]$ such that $\sigma'|_\sigma \equiv \sigma$, as well as $\sigma'(e_1) \Downarrow \sigma'(v_1)$ and, through restriction to $\mathit{dom}(\kappa_0)$, $\sigma'(e_0) \Downarrow (\mathit{rec}\ (f : T_1 \to \overline{T}_2)\ x = \sigma(e_2))$.

Using a Termination Preservation corollary for the narrowing semantics (that can be proved identically to the one for the matching semantics), in addition to Substitution Interaction as in the narrowing proof, we can use the inductive hypothesis for the substituted e' to complete the proof.

The rest of the cases follow using similar arguments, with the same overall structure as the narrowing proof. The only cases that are interestingly different (and where the termination assumption actually comes into play) are the ones that necessitate use of the combining case rule **M-Case-1**, which are **P-Case-L** and **P-Case-R**.

Case P-Case-L: Once again, the only interestingly different cases are the ones for the pattern matching constructs. For **P-Case-L**, we know that

$$e_p \Downarrow L_{\overline{T}_1+\overline{T}_2}\ v_{p_1} \text{ and } e_{p_1}[v_{p_1}/x] \Downarrow v_{p_1}'.$$

Let $(\kappa_0, [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2]$ and $\sigma_0 = \sigma \oplus u_1 \mapsto v_{p_1}$.

As usual, we can immediately use the inductive hypothesis for the predicate derivation of e to obtain κ_1, σ_1, q_1 and t_1 such that

$$\sigma_1|_{\sigma_0} \equiv \sigma_0, \quad \sigma_1 \in [\![\kappa_1]\!] \text{ and}$$
$$L_{\overline{T}_1+\overline{T}_2}\ u_1 \Leftarrow e_1 \dashv \kappa_0 \Uparrow_{q_1}^{t_1} \{\kappa_1\}.$$

However, we can't conclude that there exists a similar derivation for $R_{\overline{T}_1+\overline{T}_2}\ u_2$ from some inductive hypothesis since we don't have a corresponding derivation! That's where the termination assumptions comes in: by the Termination Lemma (Lemma A.0.0.13) there exists some $\kappa^?$ such that $R_{\overline{T}_1+\overline{T}_2}\ u_2 \Leftarrow e_1 \dashv \kappa_0 \Uparrow_{q_2}^{t_2} \kappa^?$.

We now do case analysis on $\kappa^?$. If it is equal to $\varnothing$, then the proof is straightforward following rule **M-Case-3**, using the inductive hypothesis for the other predicate derivation.

If, on the other hand, $\kappa^? = \{\kappa_2\}$ for some κ_2, we face a similar problem for the second derivation. We can obtain κ_a, σ_a, q_1' and t_1' such that

$$\sigma_a|_{\sigma_1} \equiv \sigma_1, \quad \sigma_a \in [\![\kappa_a]\!] \text{ and}$$
$$p \Leftarrow e_1[u_1/x] \dashv \kappa_1 \Uparrow_{q_1'}^{t_1'} \{\kappa_a\}$$

by the inductive hypothesis for the derivation of $e_{p_1}[v_{p_1}/x]$, but we have no corresponding derivation for the other branch. Using the Termination Lemma once again, we can obtain the there exists some such $\kappa_b^?$.

Once again we do case analysis on $\kappa_b^?$. If $\kappa_b^? = \varnothing$ then the branch of *combine* that fires returns $\{\kappa_a\}$ and the result follows directly. If $\kappa_b^? = \{\kappa_b\}$ for some κ_b, then, by the specification of *union*, σ_a is contained in the denotation of the combination and the result follows. □

Appendix B

Luck Examples

In this appendix we present the Luck programs that serve as both predicates and generators for the small examples of §4.5.

```
sig sorted :: [Int] -> Bool
fun sorted l =
    case l of
    | (x:y:t) -> x < y && sorted (y:t)
    | _ -> True
    end

sig member :: Int -> [Int] -> Bool
fun member x l =
    case l of
    | h:t -> x == h || member x t
    | _ -> False
    end
```

```
sig distinctAux :: [Int] -> [Int] -> Bool
fun distinctAux l acc =
    case l of
    | []  -> True
    | h:t -> not (member h acc) !h
             && distinctAux t (h:acc)
    end

sig distinct :: [Int] -> Bool
fun distinct l = aux l []
```

In order to obtain lists of a specific size, we could skew the distribution towards the `cons` case using numeric annotations on the branches, or, we can use the conjunction of such a predicate with the following simple length predicate (which could be greatly simplified with some syntactic sugar).

```
sig length :: [a] -> Int -> Bool
fun length l n =
    if n == 0 then
      case l of
      | [] -> True
      | _  -> False
      end
    else case l of
         | h:t -> length t (n-1)
         | _ -> False
         end
```

Finally, the Luck program that generates red black trees of a specific height is:

```
data Color = Red | Black
data RBT a = Leaf | Node Color a (RBT a) (RBT a)
```

```
fun isRBT h low high c t =
  if h == 0 then
      case (c, t) of
        | (_, Leaf) -> True
        | (Black, Node Red x Leaf Leaf) ->
          (low < x && x < high) !x
        | _ -> False
      end
  else case (c, t) of
         | (Red, Node Black x l r) ->
              (low < x && x < high) !x
              && isRBT (h-1) low x  Black l
              && isRBT (h-1) x high Black r
         | (Black, Node Red x l r) ->
              (x | low < x && x < high) !x
              && isRBT h low x  Red l
              && isRBT h x high Red r
         | (Black, Node Black x l r) ->
              (x | low < x && x < high) !x
              && isRBT (h-1) low x  Black l
              && isRBT (h-1) x high Black r
         | _ -> False
       end
```

Bibliography

[1] Sergio Antoy. A needed narrowing strategy. In *Journal of the ACM*, volume 47, pages 776–822. ACM Press, 2000.

[2] Thomas Arts, Laura M. Castro, and John Hughes. Testing Erlang data types with QuviQ QuickCheck. In *7th ACM SIGPLAN Workshop on Erlang*, pages 1–8. ACM, 2008.

[3] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Workshop on Programming Languages and Analysis for Security (PLAS)*, PLAS, pages 113–124. ACM, 2009.

[4] Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th Workshop on Programming Languages and Analysis for Security*, PLAS, pages 3:1–3:12. ACM, 2010.

[5] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with Veritesting. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1083–1094, 2014.

[6] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. In *Proceedings of the 41st Symposium on Principles of Programming Languages (POPL)*, POPL, pages 165–178. ACM, January 2014.

[7] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.

[8] Musard Balliu, Mads Dam, and Gurvan Le Guernic. Encover: Symbolic exploration for information flow security. In *25th IEEE Computer Security Foundations Symposium (CSF 2012)*, pages 30–44. IEEE, 2012.

[9] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *17th International Symposium on Formal Methods (FM)*, volume 6664 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2011.

[10] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

[11] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 14–25. ACM, 2004.

[12] Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In *2nd International Conference on Software Engineering and Formal Methods (SEFM)*, pages 230–239. IEEE Computer Society, 2004.

[13] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. Generalizing permissive-upgrade in dynamic information flow analysis. In *9th Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 15–24. ACM, 2014.

[14] Arnar Birgisson, Daniel Hedin, and Andrei Sabelfeld. Boosting the permissiveness of dynamic information-flow tracking by testing. In *17th European Symposium on Research in Computer Security*, ESORICS, pages 55–72. Springer, 2012.

[15] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *First International Conference on Interactive Theorem Proving (ITP)*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2010.

[16] Achim D. Brucker and Burkhart Wolff. Interactive testing with HOL-TestGen. In *Proceedings of the 5th International Conference on Formal Approaches to Software Testing*, FATES'05, pages 87–102, Berlin, Heidelberg, 2006. Springer-Verlag.

[17] Lukas Bulwahn. The new Quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. In *2nd International Conference on Certified*

Programs and Proofs (CPP), volume 7679 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2012.

[18] Lukas Bulwahn. Smart testing of functional programs in Isabelle. In *18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 7180 of *Lecture Notes in Computer Science*, pages 153–167. Springer, 2012.

[19] J. Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 443–446, 2008.

[20] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX conference on Operating systems design and implementation*, OSDI, pages 209–224. USENIX Association, 2008.

[21] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *13th ACM conference on Computer and communications security*, CCS '06, pages 322–335. ACM, 2006.

[22] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *33rd International Conference on Software Engineering*, ICSE '11, pages 1066–1071. ACM, 2011.

[23] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[24] Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. Constraint reasoning in FocalTest. In *5th International Conference on Software and Data Technologies*, pages 82–91. SciTePress, 2010.

[25] Arun T. Chaganty, Aditya V. Nori, and Sriram K. Rajamani. Efficiently sampling probabilistic programs via program analysis. In *Artificial Intelligence and Statistics (AISTATS)*, April 2013.

[26] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Balancing scalability and uniformity in SAT witness generator. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 60:1–60:6, New York, NY, USA, 2014. ACM.

[27] Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios. Integrating testing and interactive theorem proving. In *10th International Workshop on the ACL2 Theorem Prover and its Applications*, volume 70 of *EPTCS*, pages 4–19, 2011.

[28] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 143–156, New York, NY, USA, 2008. ACM.

[29] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[30] Jan Christiansen and Sebastian Fischer. EasyCheck – test data for free. In *9th International Symposium on Functional and Logic Programming (FLOPS)*, volume 4989 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2008.

[31] Koen Claessen, Jonas Duregård, and Michał H. Pałka. Generating constrained random data with uniform distribution. In *Functional and Logic Programming*, volume 8475 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2014.

[32] Koen Claessen, Jonas Duregård, and Michal H. Palka. Generating constrained random data with uniform distribution. *J. Funct. Program.*, 25, 2015.

[33] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279. ACM, 2000.

[34] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[35] David Delahaye, Catherine Dubois, and Jean-Frédéric Étienne. Extracting purely functional contents from logical inductive types. In *20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 4732 of *Lecture Notes in Computer Science*, pages 70–85. Springer, 2007.

[36] Maxime Dénès, Cătălin Hriţcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. QuickChick: Property-based testing for Coq. The Coq Workshop, July 2014.

[37] Jonas Duregård, Patrik Jansson, and Meng Wang. Feat: Functional enumeration of algebraic types. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 61–72, New York, NY, USA, 2012. ACM.

[38] Richard Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420–, July 1964.

[39] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Combining testing and proving in dependent type theory. In *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2758 of *Lecture Notes in Computer Science*, pages 188–203. Springer, 2003.

[40] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Verifying haskell programs by combining testing, model checking and interactive theorem proving. *Information & Software Technology*, 46(15):1011–1025, 2004.

[41] Carl Eastlund. DoubleCheck your theorems. In *ACL2*, 2009.

[42] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974.

[43] Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes, and Robert Bruce Findler. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. In *24th European Symposium on Programming*, volume 9032 of *Lecture Notes in Computer Science*, pages 383–405. Springer, 2015.

[44] Sebastian Fischer and Herbert Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 63–74. ACM, 2007.

[45] Jonathan Fowler and Graham Huttom. Towards a theory of reach. In *Trends in Functional Programming - 16th International Symposium, TFP 2015, Sophia Antipolis, France, June 3-5, 2015. Revised Selected Papers*, volume 9547 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 2015.

[46] Diane Gallois-Wong. Formalising Luck: Improved probabilistic semantics for property-based generators. Inria Internship Report, August 2016.

[47] Cale Gibbard, Brent Yorgey, et al. `MonadRandom`: Random-number generation monad. `http://hackage.haskell.org/package/MonadRandom-0.4.2.3`, April 2016.

[48] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In *32nd ACM/IEEE International Conference on Software Engineering*, pages 225–234. ACM, 2010.

[49] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 213–223. ACM, 2005.

[50] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *ACM Queue*, 10(1):20, 2012.

[51] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*, pages 220–229, 2008.

[52] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In James D. Herbsleb and Matthew B. Dwyer, editors, *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, pages 167–181. ACM, 2014.

[53] Arnaud Gotlieb. Euclide: A constraint-based testing framework for critical C programs. In *ICST 2009, Second International Conference on Software Testing Verification and Validation, 1-4 April 2009, Denver, Colorado, USA*, pages 151–160, 2009.

[54] Alex Groce, Gerard J. Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *ICSE*, pages 621–631. IEEE Computer Society, 2007.

[55] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 78–88, New York, NY, USA, 2012. ACM.

[56] Katarzyna Grygiel and Pierre Lescanne. Counting and generating lambda terms. *CoRR*, abs/1210.2610, 2012.

[57] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 653–669, 2016.

[58] Qiao Haiyan. *Testing and Proving in Dependent Type Theory*. PhD thesis, Chalmers, 2003.

[59] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[60] Michael Hanus. A unified computation model for functional and logic programming. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 80–93. ACM Press, 1997.

[61] Haskell Libraries Team and Roman Leshchinskiy. `vector`: Efficient arrays. `http://hackage.haskell.org/package/vector-0.11.0.0`, December 2015.

[62] Daniel Hedin and Andrei Sabelfeld. Information-flow security for a core of JavaScript. In *25th IEEE Computer Security Foundations Symposium (CSF)*, CSF, pages 3–18. IEEE, 2012.

[63] Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill$^{\text{tm}}$: A bayesian skill rating system. In *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*, pages 569–576, 2006.

[64] Cătălin Hriţcu, Michael Greenberg, Ben Karel, Benjamin C. Pierce, and Greg Morrisett. All your IFCException are belong to us. In *34th IEEE Symposium on Security and Privacy*, pages 3–17. IEEE Computer Society Press, May 2013.

[65] Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. Testing noninterference, quickly. In *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 455–468. ACM, 2013.

[66] Cătălin Hriţcu, Leonidas Lampropoulos, Antal Spector-Zabusky, Arthur Azevedo de Amorim, Maxime Dénès, John Hughes, Benjamin C. Pierce, and Dimitrios Vytiniotis. Testing noninterference, quickly. *Journal of Functional Programming (JFP); Special issue for ICFP 2013*, 26:e4 (62 pages), April 2016. Technical Report available as arXiv:1409.0393.

[67] John Hughes. QuickCheck testing for fun and profit. In *9th International Symposium on Practical Aspects of Declarative Languages (PADL)*, volume 4354 of *Lecture Notes in Computer Science*, pages 1–32. Springer, 2007.

[68] Daniel Jackson. *Software Abstractions: Logic, Language, and Anlysis*. The MIT Press, 2011.

[69] Thomas P. Jensen. Disjunctive program analysis for algebraic data types. *ACM Trans. Program. Lang. Syst.*, 19(5):751–803, 1997.

[70] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.

[71] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[72] Gowtham Kaki and Suresh Jagannathan. A relational framework for higher-order shape analysis. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 311–324. ACM, 2014.

[73] Andrew J. Kennedy and Dimitrios Vytiniotis. Every bit counts: The binary representation of typed data and programs. *Journal of Functional Programming*, 22(4-5):529–573, 2012.

[74] Johannes Kinder. Hypertesting: The case for automated testing of hyperproperties. 3rd Workshop on Hot Issues in Security Principles and Trust (HotSpot), 2015.

[75] Casey Klein. Experience with randomized testing in programming language metatheory. Master's thesis, Northwestern, August 2009. `http://plt.eecs.northwestern.edu/klein-masters.pdf`.

[76] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: On the effectiveness of lightweight mechanization. In *Principles of Programming Languages (POPL)*, 2012.

[77] Casey Klein and Robert Bruce Findler. Randomized testing in PLT Redex. In *Workshop on Scheme and Functional Programming (SFP)*, 2009.

[78] Casey Klein, Matthew Flatt, and RobertBruce Findler. The Racket virtual machine and randomized testing. *Higher-Order and Symbolic Computation*, pages 1–45, 2013.

[79] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Scala to the power of Z3: integrating SMT and programming. In *23rd International Conference on Automated Deduction*, volume 6803 of *Lecture Notes in Computer Science*, pages 400–406. Springer, 2011.

[80] Pieter W. M. Koopman, Peter Achten, and Rinus Plasmeijer. Model-based shrinking for state-based testing. In *14th International Symposium on Trends in Functional Programming (TFP)*, pages 107–124, 2013.

[81] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. Beginner's Luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 114–129, 2017.

[82] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *Proc. ACM Program. Lang.*, 2(POPL):45:1–45:30, December 2017.

[83] Leonidas Lampropoulos, Antal Spector-Zabusky, and Kenneth Foner. Ode on a random urn (functional pearl). In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell 2017, pages 26–37, New York, NY, USA, 2017. ACM.

[84] Krzysztof Łatuszyński, Gareth O. Roberts, and Jeffrey S. Rosenthal. Adaptive gibbs samplers and related mcmc methods. *The Annals of Applied Probability*, 23(1):66–98, 02 2013.

[85] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[86] Pierre Lescanne. Boltzmann samplers for random generation of lambda terms. *CoRR*, abs/1404.3875, 2014.

[87] Fredrik Lindblad. Property directed generation of first-order test data. In *8th Symposium on Trends in Functional Programming*, volume 8 of *Trends in Functional Programming*, pages 105–123. Intellect, 2007.

[88] Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1):99–118, 1977.

[89] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th international conference on Software Engineering*, ICSE, pages 416–426. IEEE Computer Society, 2007.

[90] Vikash K. Mansinghka, Daniel M. Roy, Eric Jonas, and Joshua B. Tenenbaum. Exact and approximate sampling by systematic stochastic search. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics, AISTATS 2009, Clearwater Beach, Florida, USA, April 16-18, 2009*, pages 400–407, 2009.

[91] Brian Milch, Bhaskara Marthi, Stuart J. Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: probabilistic models with unknown objects. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, pages 1352–1359, 2005.

[92] Dimiter Milushev, Wim Beck, and Dave Clarke. Noninterference via symbolic execution. In *FMOODS/FORTE*, volume 7273 of *Lecture Notes in Computer Science*, pages 152–168. Springer, 2012.

[93] Flemming Nielson and Hanne Riis Nielson. Tensor products generalize the relational data flow analysis method. In *4th Hungarian Computer Science Conference*, pages 211–225, 1985.

[94] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[95] Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. R2: An efficient mcmc sampler for probabilistic programs. In *AAAI Conference on Artificial Intelligence (AAAI)*. AAAI, July 2014.

[96] Martín Ochoa, Jorge Cuéllar, Alexander Pretschner, and Per Hallgren. Idea: Unwinding based model-checking and testing for non-interference on EFSMs. In *7th International Symposium on Engineering Secure Software and Systems (ESSoS)*, volume 8978 of *Lecture Notes in Computer Science*, pages 34–42. Springer, 2015.

[97] Chris Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, 1999.

[98] Bryan O'Sullivan. Criterion: a Haskell microbenchmarking library. `http://www.serpentine.com/criterion/`, 2014.

[99] Sam Owre. Random testing in PVS. In *Workshop on Automated Formal Methods*, 2006.

[100] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for Java. In *22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems And Applications*, OOPSLA, pages 815–816. ACM, 2007.

[101] Michał H. Pałka. Testing an optimising compiler by generating random lambda terms. `http://www.cse.chalmers.se/~palka/testingcompiler/`.

[102] Michał H. Pałka. *Random Structured Test Data Generation for Black-Box Testing*. Doktorsavhandlingar vid Chalmers tekniska hgskola. Ny serie, no:. Department of Computer Science and Engineering, Software Technology (Chalmers), Chalmers University of Technology,, 2014. 168.

[103] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 91–97, New York, NY, USA, 2011. ACM.

[104] Manolis Papadakis and Konstantinos F. Sagonas. A proper integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang, Tokyo, Japan, September 23, 2011*, pages 39–50, 2011.

[105] Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In Christian Urban and Xingyuan Zhang, editors, *6th International Conference on Interactive Theorem Proving (ITP)*, volume 9236 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2015.

[106] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, Version 4.0 beta, January 2016.

[107] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.

[108] QuickCheck developers, Nick Smallbone, Bjrn Bringert, and Koen Claessen. QuickCheck: Automatic testing of Haskell programs. `http://hackage.haskell.org/package/QuickCheck-2.8.2`, January 2016.

[109] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 335–346. ACM, 2012.

[110] Jason S. Reich, Matthew Naylor, and Colin Runciman. Lazy generation of canonical test programs. In *23rd International Symposium on Implementation and Application of Functional Languages*, volume 7257 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 2011.

[111] Alexey Rodriguez Yakushev and Johan Jeuring. Enumerating well-typed terms generically. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Approaches and Applications of Inductive Programming*, volume 5812 of *Lecture Notes in Computer Science*, pages 93–116. Springer Berlin Heidelberg, 2010.

[112] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In *1st ACM SIGPLAN Symposium on Haskell*, pages 37–48. ACM, 2008.

[113] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *23rd Computer Security Foundations Symposium (CSF)*, CSF, pages 186–199. IEEE Computer Society, 2010.

[114] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[115] Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Ershov Memorial Conference*, pages 352–365. Springer, 2009.

[116] Eric L. Seidel, Niki Vazou, and Ranjit Jhala. Type targeted testing. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 812–836, 2015.

[117] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272. ACM, 2005.

[118] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *4th Symposium on Haskell*, pages 95–106. ACM, 2011.

[119] Dominic Steinitz and James Cook. `random-fu`: Random number generation. `http://hackage.haskell.org/package/random-fu-0.2.6.2`, January 2015.

[120] Paul Tarau. On type-directed generation of lambda terms. In *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), Cork, Ireland, August 31 - September 4, 2015.*, 2015.

[121] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In *12th International Symposium on Static Analysis (SAS)*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2005.

[122] Pierre-Nicolas Tollitte, David Delahaye, and Catherine Dubois. Producing certified functional code from inductive specifications. In *Second International Conference on Certified Programs and Proofs (CPP)*, volume 7679 of *Lecture Notes in Computer Science*. Springer, 2012.

[123] Andrew P. Tolmach and Sergio Antoy. A monadic semantics for core Curry. *Electr. Notes Theor. Comput. Sci.*, 86(3):16–34, 2003.

[124] Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 54. ACM, 2014.

[125] Nicky Williams, Bruno Marre, and Patricia Mouy. On-the-fly generation of K-path tests for C functions. In *19th IEEE International Conference on Automated Software Engineering*, ASE, pages 290–293. IEEE, 2004.

[126] Sean Wilson. *Supporting dependently typed functional programming with proof automation and testing*. PhD thesis, The University of Edinburgh, June 2011.

[127] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294, 2011.

[128] Stephan A. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, August 2002.

[129] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[130] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2-3):67–84, 2007.