UNIVERSITY OF CALIFORNIA, SAN DIEGO

Liquid Haskell: Haskell as a Theorem Prover

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Niki Vazou

Committee in charge:

Professor Ranjit Jhala, Chair
Professor Samuel R. Buss
Professor Cormac Flanagan
Professor Sorin Lerner
Professor Daniele Micciancio

2016

The Dissertation of Niki Vazou is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2016

DEDICATION

For my mother, father, and sister.

EPIGRAPH

Simplicity is the ultimate sophistication.

Leonardo Da Vinci

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I want to thank Ranjit Jhala for being such a great supervisor. It is his unique way to interpret, simplify, and beautify my thoughts and his strong willingness and enthusiasm to do useful work that directed my research and influenced the way I am thinking.

A big thanks to my committee members Sam Buss, Cormac Flanagan, Sorin Lerner, and Daniele Micciancio for their interest and insights in my work.

Next, I want to thank all my labmates in the UCSD programming languages group and specifically my collaborators Eric Seidel, Alexander Bakst, Pat Rondon, and Valentin Robert for providing a supportive, friendly, and inspiring working environment.

I own a big thanks all these people who hosted me in my internships. The Opa group in Paris who foresaw that types can help industrial development. Dimitrios Vytiniotis and Simon Peyton-Jones at Microsoft Research Cambridge who were the first Haskellers who believed in LIQUID HASKELL and have been still supporting both the project and me personally. I want to thank Jeff Polakow, Gabriel Gonzalez, and all the people at Awake Networks for giving me the opportunity to spend an awesome summer in Mountain View and use LIQUID HASKELL in real world development code. A huge thanks to the two main people responsible for my great summer at Microsoft Research Redmond. Rustan Leino who is the innovator in automatic software verification and yet is always open to discuss, in his unique friendly and humble manner, how Dafny and LIQUID HASKELL could be further improved and influenced from one another. Last but not least, I want to thank my amazing internship host Daan Leijen for mentoring me during my internship and since then, sharing with me all his insightful knowledge and experience about research and life in general.

Thanks to all the Haskell community for the enthusiasm and support expressed for LIQUID HASKELL from its earliest stages. I am so proud to be a member of this supportive, friendly, and respectful community! I want to thank those who introduced me to Haskell and Computer Science in general, Nikolaos Papaspyrou and Stathis Zachos.

Next, I want to thank all my San Diego friends who were next to me during times of both joy and sorrow and who turned this beautiful place into my second home. Sincere thanks to all

my women friends Sohini, Minu, Augusta, Margherita, Marta, and Karyn who kept me sane and happy during my male dominated Ph.D.. I want to thank Andreas, Dimos, Vasilis, Konstantinos, Andreas and the rest of the Greeks in San Diego who help me keep my culture and roots while so far away from my country. Finally, a great thanks to the crew and friends in Pappalecco, the coffee place where most of the LIQUID HASKELL code and papers were written in.

Last but not least, I want to thank my family and friends, around the world, who supported me throughout this journey. All the people I met in my numerous trips in these five years, that made me realize how knowledge unites people from different parts of the globe. All my friends and family in Greece that despite the distance, always welcome me during my escapes in my home country. Specifically, my parents Voula and Nikos and my sister Marianna who stayed up all these Sunday nights for our online calls.

It is great that the destination is reached, but after all, it is the journey that matters.

Niki Vazou

December 2016

Chapter 1 contains material adapted from the following publication: N. Vazou, E. Seidel, and R. Jhala, "LiquidHaskell: Experience with Refinement Types in the Real World", Haskell, 2014.

Chapter 2 contains material adapted from the following publication: N. Vazou, E. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement Types for Haskell", ICFP, 2014.

Chapter 3 contains material adapted from the following publication: N. Vazou, P. Rondon, and R. Jhala, "Abstract Refinement Types", ESOP, 2013.

Chapter 4 contains material adapted from the following publication: N. Vazou, A. Bakst, and R. Jhala, "Bounded Refinement Types", ICFP, 2015.

Chapter 5 has been submitted for publication of the material as it may appear in PLDI 2017: Vazou, Niki; Choudhury, Vikraman; Scott, Ryan G.; Newton, Ryan R.; Jhala, Ranjit. "Refinement Reflection: Parallel Legacy Languages as Theorem Provers".

Chapter 6 has been submitted for publication of the material as it may appear in ESOP 2017: Vazou, Niki; Polakow, Jeff. "Verified Parallel String Matching in Haskell".

The dissertation author was the primary investigator and author of these papers.

VITA

2010 Diploma in Computer Software, National Technical University of Athens

2016 Ph.D. in Computer Science, University of California, San Diego

PUBLICATIONS

N. Vazou, M. Papakyriakou, and N. Papaspyrou, "Memory Safety and Race Freedom in Concurrent Programming with Linear Capabilities", FedCSIS, 2011.

N. Vazou, P. Rondon, and R. Jhala, "Abstract Refinement Types", ESOP, 2013.

N. Vazou, E. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement Types for Haskell", ICFP, 2014.

N. Vazou, E. Seidel, and R. Jhala, "LiquidHaskell: Experience with Refinement Types in the Real World", Haskell, 2014.

E. Seidel, N. Vazou, and R. Jhala, "Type Targeted Testing", ESOP, 2015.

N. Vazou, A. Bakst, and R. Jhala, "Bounded Refinement Types", ICFP, 2015.

N. Vazou, and D. Leijen, "From Monads to Effects and Back", PADL 2016.

ABSTRACT OF THE DISSERTATION

Liquid Haskell: Haskell as a Theorem Prover

by

Niki Vazou

Doctor of Philosophy in Computer Science

University of California, San Diego, 2016

Professor Ranjit Jhala, Chair

Code deficiencies and bugs constitute an unavoidable part of software systems. In safety-critical systems, like aircrafts or medical equipment, even a single bug can lead to catastrophic impacts such as injuries or death. Formal verification can be used to statically track code deficiencies by proving or disproving correctness properties of a system. However, at its current state formal verification is a cumbersome process that is rarely used by mainstream developers, mostly because it targets non general purpose languages (e.g., Coq, Agda, Dafny).

We present LIQUID HASKELL, a *usable* program verifier that aims to establish formal verification as an integral part of the development process. LIQUID HASKELL *naturally integrates* the specification of correctness properties as logical refinements of Haskell's types. Moreover, it

uses the abstract interpretation framework of liquid types to *automatically* check correctness of specifications via Satisfiability Modulo Theories (SMT) solvers requiring no explicit proofs or complicated annotations. Finally, the specification language is arbitrary *expressive*, allowing the user to write general correctness properties about their code, thus turning Haskell into a theorem prover.

Transforming a mature language — with optimized libraries and highly tuned parallelism — into a theorem prover enables us to verify a wide variety of properties on real world applications. We used LIQUID HASKELL to verify shallow invariants of existing Haskell code, *e.g.* memory safety of the optimized string manipulation library `ByteString`. Moreover, we checked deep, sophisticated properties of parallel Haskell code, *e.g.* program equivalence of a naïve string matcher and its parallelized version. Having verified about 20K of Haskell code, we present how LIQUID HASKELL serves as a prototype verifier in a future where formal techniques will be used to facilitate, instead of hinder, software development.

Introduction

Code deficiencies and bugs constitute an unavoidable part of software systems. In safety-critical systems, like aircrafts or medical equipment, even a single bug can lead to catastrophic impacts such as injuries or death. Even in less critical code, programs use various runtime assertions to establish correctness properties. Formal verification can be used to statically discharge assertions and track code deficiencies by proving or disproving correctness properties of a system. However, at its current state formal verification is a cumbersome process that is rarely used by mainstream developers.

We present LIQUID HASKELL, a *usable* program verifier that aims to establish formal verification as an integral part of the development process. A usable verifier *naturally integrates* the specification of correctness properties in the development process. Moreover, verification should be *automatic*, requiring no explicit proofs or complicated annotations. At the same time, the specification language should be *expressive*, allowing the user to write arbitrary correctness properties. Finally, a usable verifier should be tested in *real-world programs*.

LIQUID HASKELL is a verifier for Haskell programs that takes as input Haskell source code, annotated with correctness specifications in the form of refinement types and checks whether the code satisfies the specifications. We designed LIQUID HASKELL in such a way as to satisfy most of the aforementioned criteria of a usable verifier.

Natural Integration of correctness specifications comes by our choice of the functional programming language Haskell as a target language. Haskell's first class functions lead to modular specifications. The lack of mutations and side-effects allows a direct correspondence between source code and logic. Correctness specifications are naturally added to Haskell's expressive type system as *refinement types*, *i.e.* types annotated with logical predicates. As an example, the type

```
type NonZero = {v:Int | 0 ≠ v}
```

describes a value `v` which is an integer and the refinement specifies that this value is not zero. The specification language is simple as most programmers are familiar with both its ingredients, *i.e.* Haskell types and logical formulas.

Real World Applications have been verified using LIQUID HASKELL. We proved critical safety and functional correctness of more that 10K lines of popular Haskell libraries (Chapter 1) with minimal amount of annotations. We verified correctness of *array-based sorting algorithm* (`Vector-Algorithms`), preservation of *binary search tree* properties (`Data.Map`, `Data.Set`), preservation of *uniqueness invariants* (`XMonad`), *low-level memory safety* (`Bytestring`, `Text`), and even found and fixed a subtle correctness bug related to unicode handling in `Text`. In the above libraries we automatically proved *totality* and *termination* of all interface functions. Even though most of Haskell's features facilitate verification, lazy semantics rendered standard refinement typing unsound.

Soundness Under Lazy Evaluation (Chapter 2) describes how we adjusted refinement typing to soundly verify Haskell's lazy programs. Refinement types were introduced in 1991 and since then have been successfully applied to many eager languages. When checking an expression, such type systems implicitly assume that all the free variables in the expression are bound to values. This property is trivially guaranteed by eager evaluation, but does not hold in a lazy setting. Thus, to be sound and precise, a refinement type system for Haskell must take into account which subset of binders actually reduces to values. To track potentially diverging binders, we built a termination checker whose correctness is recursively checked by refinement types.

Automatic Verification comes by constraining refinements in specifications to decidable logics. Program verification checks that the source code satisfies a set of specifications. A trivial example is to *specify* that the second argument of a division operator is different than zero, by writing the following specification: `div :: Int → NonZero → Int`. To *check* whether an expression with type `{v:Int | 0 < v}` is a safe argument to the division operator, the system checks whether $0 < v$ implies $0 \neq v$. By constraining all predicates to be drawn from decidable logics, such implications can be *automatically* checked via an Satisfiability Modulo Theories

(SMT) solver. *Liquid Types* [47] are a subset of refinement types that achieve automation and type inference by constraining the language of the logical predicates to quantifier-free *decidable* logics, including logical formulas, linear arithmetic and uninterpreted functions.

Expressiveness of the specifications is critically hindered by our choice to constrain the language of predicates to decidable logics. Liquid types specifications are naturally used to describe first order properties but prevent modular, higher order specifications. Consider a function that sorts lists of integers, with type `sort :: [Int] → [Int]`. Using LIQUID HASKELL we can specify that sorting positive numbers returns a list of positive numbers, but we cannot give a modular specification accounting for *all* different kinds of numbers `sort` will be invoked. We developed "Abstract" and "Bounded" refinement types to allow for modular specifications while preserving SMT decidability.

In **Abstract Refinement Types** (Chapter 3) we parameterize a type over its refinements allowing modular specifications while preserving SMT-based decidable type checking. As an example, since `sort` preserves the elements of the input list, we can use abstract refinements to specify that *for every* refinement p on integers, `sort` takes a list of integers that satisfy p and returns a list of integers that satisfy the same refinement p.

```
sort :: ∀ <p :: Int → Bool>. [{v:Int | p v}] → [{v:Int | p v}]
```

With this modular specification, we can prove that `sort` preserves the property that all the input numbers satisfy, for any property, ranging from being positive numbers to being numbers that are safe keys for a security protocol. We used abstract refinements to describe modular properties of recursive data structures. With such abstractions we simultaneously reasoned about challenging invariants such as *sortedness and uniqueness of lists* or preservation of *red-black invariants or heap properties* on trees. Without abstract refinements reasoning about each of these invariants would require a special purpose analysis. Crucially, abstractions over refinements preserve SMT-based decidability, simply by encoding refinement parameters as uninterpreted propositions within the ground refinement logic.

Bounded Refinement Types (Chapter 4) constrain and relate abstract refinement and let us express even more interesting invariants while preserving SMT-decidability. As an example, we

used bounds on refinement types to reason about *stateful computations*. We expressed the pre- and post-conditions of the computations with two abstract refinements, p and q respectively and used bounds to impose constraints upon them. For instance, when sequencing two computation we bound the first post-condition q_1 to imply the second pre-condition p_2. We implemented the above idea in a refined Haskell `IO` state monad that encodes *Floyd-Hoare logic state transformations* and used this encoding to track capabilities and resource usage. Moreover, we encoded *safe database access* using abstract refinements to encode key-value properties and bounds to express the constraints imposed by relational algebra operators, like disjointedness, union *etc.*. Bounds are internally translated to "ghost" functions, thus the increased expressiveness comes while preserving the automated and decidable SMT-based type checking that makes liquid typing effective in practice. Abstract and Bounded refinement types do allow modular higher order specifications, but the expressiveness of the specifications is crucially restricted by the fact that, for automatic verification, arbitrary, Haskell functions are not allowed to appear in the refinements.

Refinement Reflection (Chapter 5) allows arbitrary, terminating, Haskell functions to appear into the specifications as uninterpreted functions thus preserving automatic and decidable type checking. The key idea is to reflect the code implementing a user-defined function into the function's (output) refinement type. As a consequence, at *uses* of the function, the function definition is unfolded into the refinement logic in a precise and predictable manner. With Refinement Reflection, the user can write arbitrarily expressive (fully dependent type) specifications expressing theorems about the code, but to prove such theorems the user needs to manually provide appropriate proof terms. We used reflection to verify that many widely used instances of the Monoid, Applicative, Functor and Monad typeclasses satisfy key algebraic laws needed to making the code using the typeclasses safe. Finally, transforming a mature language—with highly tuned parallel runtime—into a theorem prover enables us to build parallel applications, like an efficient String Matcher (Chapter 6), and prove it equivalent with its naïve, sequential version.

In short, LIQUID HASKELL is a usable verifier for real world Haskell applications as it allows for natural integration of expressive, type based specifications that can be automatically verified using SMT solvers.

Chapter 1

Refinement Types in Practice

Everything should be made as simple as possible, but no simpler.

– Albert Einstein

Refinement types enable specification of complex invariants by extending the base type system with *refinement predicates* drawn from decidable logics. For example,

```
type Nat = {v:Int | 0 ≤ v}
type Pos = {v:Int | 0 < v}
```

are refinements of the basic type `Int` with a logical predicate that states the *values* `v` being described must be *non-negative* and *postive* respectively. We can specify *contracts* of functions by refining function types. For example, the contract for `div`

```
div :: n:Nat → d:Pos → {v:Nat | v ≤ n}
```

states that `div` *requires* a non-negative dividend `n` and a positive divisor `d` and *ensures* that the result is less than the dividend. If a program (refinement) type checks, we can be sure that `div` will never throw a divide-by-zero exception.

Refinement types [20, 81] have been implemented for several languages like ML [106, 7, 79], C [19, 80], TypeScript [102], Racket [49] and Scala [82]. Here we present LIQUID HASKELL, a refinement type checker for Haskell. In this chapter we start with an example driven informal and practical overview of LIQUID HASKELL. In particular, we try to answer the following questions:

1. What properties can be specified with refinement types?

2. What inputs are provided and what feedback is received?

3. What is the process for modularly verifying a library?

4. What are the limitations of refinement types?

We attempt to investigate these questions, by using the refinement type checker LIQUID HASKELL, to specify and verify a variety of properties of over 10,000 lines of Haskell code from popular libraries, including `containers`, `hscolor`, `bytestring`, `text`, `vector-algorithms` and `xmonad`.

- First (§ 1.1), we present a high-level overview of LIQUID HASKELL, through a tour of its features.

- Second, we present a qualitative discussion of the kinds of properties that can be checked – ranging from generic application independent criteria like totality (§ 1.2), *i.e.* that a function is defined for all inputs (of a given type) and termination, (§ 1.3) *i.e.* that a recursive function cannot diverge, to application specific concerns like memory safety (§ 1.4) and functional correctness properties (§ 1.5).

- Finally (§ 1.6), we present a quantitative evaluation of the approach, with a view towards measuring the efficiency and programmer's effort required for verification, and we discuss various limitations of the approach which could provide avenues for further work.

1.1 LIQUID HASKELL

We start with a short description of the LIQUID HASKELL workflow, summarized in Figure 1.1 and continue with an example driven overview of how properties are specified and verified using the tool.

Source LIQUID HASKELL can be run from the command-line[1] or within a web-browser[2]. It

[1] https://hackage.haskell.org/package/liquidhaskell
[2] http://goto.ucsd.edu/liquid/haskell/demo/

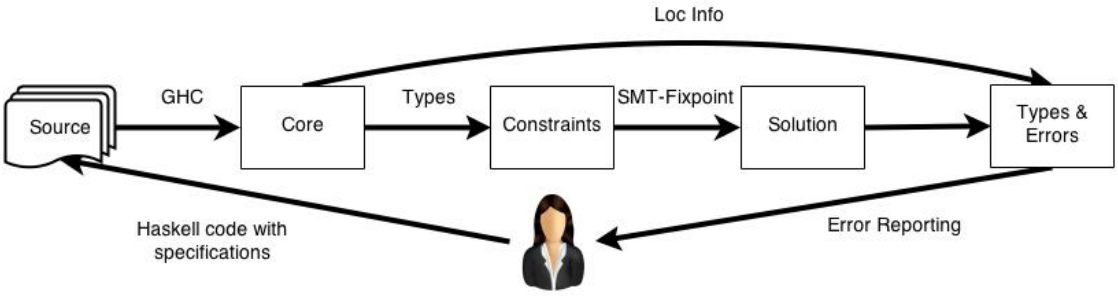

Figure 1.1. LIQUID HASKELL Workflow.

takes as *input*: (1) a single Haskell *source* file with code and refinement type specifications including refined datatype definitions, measures (§ 1.1.3), predicate and type aliases, and function signatures; (2) a set of directories containing *imported modules* (including the `Prelude`) which may themselves contain specifications for exported types and functions; and (3) a set of predicate fragments called *qualifiers*, which are used to infer refinement types. This set is typically empty as the default set of qualifiers extracted from the type specifications suffices for inference.

Core LIQUID HASKELL uses GHC to reduce the source to the Core IL [87] and, to facilitate source-level error reporting, creates a map from Core expressions to locations in the Haskell source.

Constraints Then, it uses the abstract interpretation framework of Liquid Typing [79], modified to ensure soundness under lazy evaluation 2 and extended with Abstract 3 and Bounded 4 Refinement Types and Refinement Reflection 5, to generate logical constraints from the Core IL.

Solution Next, it uses a fixpoint algorithm (from [79]) combined with an SMT solver to solve the constraints, and hence infers a valid refinement typing for the program. LIQUID HASKELL can use any solver that implements the SMT-LIB2 standard [4], including Z3 [24], CVC4 [3], and MathSat [11].

Types & Errors If the set of constraints is satisfiable, then LIQUID HASKELL outputs SAFE, meaning the program is verified. If instead, the set of constraints is not satisfiable, then LIQUID HASKELL outputs UNSAFE, and uses the invalid constraints to report refinement type errors at the *source positions* that created the invalid constraints, using the location information to map the

invalid constraints to source positions. In either case, LIQUID HASKELL produces as output a source map containing the *inferred* types for each program expression, which, in our experience, is crucial for debugging the code and the specifications.

LIQUID HASKELL is best thought of as an *optional* type checker for Haskell. By optional we mean that the refinements have *no* influence on the dynamic semantics, which makes it easy to apply LIQUID HASKELL to *existing* libraries. To emphasize the optional nature of refinements and preserve compatibility with existing compilers, all specifications appear within comments of the form `{-@ ... @-}`, which we omit below for brevity.

1.1.1 Specifications

A refinement type is a Haskell type where each component of the type is decorated with a predicate from a (decidable) refinement logic. We use the quantifier-free logic of equality, uninterpreted functions and linear arithmetic (QF-EUFLIA) [69]. For example,

```
{v:Int | 0 ≤ v ∧ v < 100}
```

describes `Int` values between `0` and `100`.

Type Aliases For brevity and readability, it is often convenient to define abbreviations for particular refinement predicates and types. For example, we can define an alias for the above predicate

```
predicate Btwn Lo N Hi = Lo ≤ N ∧ N < Hi
```

and use it to define a *type alias*

```
type Rng Lo Hi = {v:Int | Btwn Lo v Hi}
```

We can now describe the above integers as `(Rng 0 100)`.

Contracts To describe the desired properties of a function, we need simply refine the input and output types with predicates that respectively capture suitable pre- and post-conditions. For example,

```
range :: lo:Int → hi:{Int | lo ≤ hi} → [(Rng lo hi)]
```

states that `range` is a function that takes two `Int`s respectively named `lo` and `hi` and returns a list of `Int`s between `lo` and `hi`. There are three things worth noting. First, we have binders to

name the function's *inputs* (*e.g.* `lo` and `hi`) and can use the binders inside the function's *output*. Second, the refinement in the *input* type describes the *pre-condition* that the second parameter `hi` cannot be smaller than the first `lo`. Third, the refinement in the *output* type describes the *post-condition* that all returned elements are between the bounds of `lo` and `hi`.

1.1.2 Verification

Next, consider the following implementation for `range`:

```
range lo hi
  | lo ≤ hi  = lo : range (lo + 1) hi
  | otherwise = []
```

When we run LIQUID HASKELL on the above code, it reports an error at the definition of `range`. This is unpleasant! One way to debug the error is to determine what type has been *inferred* for `range`, *e.g.* by hovering the mouse over the identifier in the web interface. In this case, we see that the output type is essentially:

```
[{v:Int | lo ≤ v ∧ v ≤ hi}]
```

which indicates the problem. There is an *off-by-one* error due to the problematic guard. If we replace the second $\leq$ with a `<` and re-run the checker, the function is verified.

Holes It is often cumbersome to specify the Haskell types, as those can be gleaned from the regular type signatures or via GHC's inference. Thus, LIQUID HASKELL allows the user to leave holes in the specifications. Suppose `rangeFind` has type

```
(Int → Bool) → Int → Int → Maybe Int
```

where the second and third parameters define a range. We can give `rangeFind` a refined specification:

```
_ → lo:_ → hi:{Int | lo ≤ hi} → Maybe (Rng lo hi)
```

where the `_` is the unrefined Haskell type for the corresponding position in the type.

Inference Next, consider the implementation

```
rangeFind f lo hi = find f $ range lo hi
```

where `find` from `Data.List` has the (unrefined) type

```
find :: (a → Bool) → [a] → Maybe a
```

LIQUID HASKELL uses the abstract interpretation framework of Liquid Typing [79] to infer that the type parameter `a` of `find` can be instantiated with `(Rng lo hi)` thereby enabling the automatic verification of `rangeFind`.

Inference is crucial for automatically synthesizing types for polymorphic instantiation sites – note there is another instantiation required at the use of the apply operator `$` – and to relieve the programmer of the tedium of specifying signatures for all functions. Of course, for functions exported by the module, we must write signatures to specify preconditions – otherwise, the system defaults to using the trivial (unrefined) Haskell type as the signature *i.e.*, checks the implementation assuming arbitrary inputs.

1.1.3 Measures

So far, the specifications have been limited to comparisons and arithmetic operations on primitive values. We use *measure functions*, or just measures, to specify *inductive properties* of algebraic data types. For example, we define a measure `len` to write properties about the number of elements in a list.

```
measure len :: [a] → Int
len []     = 0
len (x:xs)  = 1 + (len xs)
```

Measure definitions are *not* arbitrary Haskell code but a very restricted subset 2.1.6. Each measure has a single equation per constructor that defines the value of the measure for that constructor. The right-hand side of the equation is a term in the restricted refinement logic. Measures are interpreted by generating refinement types for the corresponding data constructors. For example, from the above, LIQUID HASKELL derives the following types for the list data constructors:

```
[]  :: {v:[a]| len v = 0}
(:) :: _ → xs:_ → {v:[a]| len v = 1 + len xs}
```

Here, `len` is an *uninterpreted function* in the refinement logic. We can define multiple measures for a type; LIQUID HASKELL simply conjoins the individual refinements arising from each measure to obtain a single refined signature for each data constructor.

Using Measures We use measures to write specifications about algebraic types. For example, we can specify and verify that:

```
append :: xs:[a] → ys:[a]
       → {v:[a]| len v = len xs + len ys}

map    :: (a → b) → xs:[a]
       → {v:[b]| len v = len xs}

filter :: (a → Bool) → xs:[a]
       → {v:[a]| len v ≤ len xs}
```

Propositions Measures can be used to encode sophisticated invariants about algebraic data types. To this end, the user can write a measure whose output has a special type `Prop` denoting propositions in the refinement logic. For instance, we can describe a list that contains a `0` as:

```
measure hasZero :: [Int] → Prop
hasZero []      = false
hasZero (x:xs)  = x == 0 || hasZero xs
```

We can then define lists containing a `0` as:

```
type HasZero = {v : [Int] | hasZero v }
```

Using the above, LIQUID HASKELL will accept

```
xs0 :: HasZero
xs0 = [2,1,0,-1,-2]
```

but will reject

```
xs′ :: HasZero
xs′ = [3,2,1]
```

1.1.4 Refined Data Types

Often, we require that *every* instance of a type satisfies some invariants. For example, consider a `CSV` data type, that represents tables:

```
data CSV a = CSV { cols :: [String]
                 , rows :: [[a]]    }
```

With LIQUID HASKELL we can enforce the invariant that every row in a `CSV` table should have the same number of columns as there are in the header

```
data CSV a = CSV { cols :: [String]
                 , rows :: [ListL a cols] }
```

using the alias

```
type ListL a X = {v:[a]| len v = len X}
```

A refined data definition is *global* in that LIQUID HASKELL will reject any `CSV`-typed expression that does not respect the refined definition. For example, both of the below

```
goodCSV = CSV [  "Month", "Days"]
              [ ["Jan"  , "31"]
              , ["Feb   , "28"]
              , ["Mar"  , "31"] ]

badCSV  = CSV [  "Month", "Days"]
              [ ["Jan"  , "31"]
              , ["Feb   , "28"]
              , ["Mar"        ] ]
```

are well-typed Haskell, but the latter is rejected by LIQUID HASKELL. Like measures, the global invariants are enforced by refining the constructors' types.

1.1.5 Refined Type Classes

Next, let us see how LIQUID HASKELL allows verification of programs that use ad-hoc polymorphism via type classes. While the implementation of each typeclass instance is different, there is often a common interface that all instances should satisfy.

Class Measures As an example, consider the class definition

```
class Indexable f where
  size :: f a → Int
  at   :: f a → Int → a
```

For safe access, we might require that `at`'s second parameter is bounded by the `size` of the container. To this end, we define a *type-indexed* measure, using the `class measure` keyword

```
class measure sz :: a → Nat
```

Now, we can specify the safe-access precondition independent of the particular instances of `Indexable`:

```
class Indexable f where
  size :: xs:_ → {v:Nat | v = sz xs}
  at   :: xs:_ → {v:Nat | v < sz xs} → a
```

Instance Measures For each concrete type that instantiates a class, we require a corresponding definition for the measure. For example, to define lists as an instance of `Indexable`, we require the definition of the `sz` instance for lists:

```
instance measure sz :: [a] → Nat
  sz []     = 0
  sz (x:xs) = 1 + (sz xs)
```

Class measures work just like regular measures in that the above definition is used to refine the types of the list data constructors. After defining the measure, we can define the type instance as:

```
instance Indexable [] where
  size []        = 0
  size (x:xs)    = 1 + size xs

  (x:xs) `at` 0  = x
  (x:xs) `at` i  = index xs (i-1)
```

LIQUID HASKELL uses the definition of `sz` for lists to check that `size` and `at` satisfy the refined class specifications.

Client Verification At the clients of a type-class we use the refined types of class methods. Consider a client of `Indexables`:

```
sum :: (Indexable f) ⇒ f Int → Int
sum xs = go 0
  where
    go i | i < size xs = xs `at` i + go (i+1)
         | otherwise   = 0
```

LIQUID HASKELL proves that each call to `at` is safe, by using the refined class specifications of `Indexable`. Specifically, each call to `at` is guarded by a check `i < size xs` and `i` is increasing from 0, so LIQUID HASKELL proves that `xs `at` i` will always be safe.

1.2 Totality

Well typed Haskell code can go very wrong:

```
*** Exception: Prelude.head: empty list
```

As our first application, let us see how to use LIQUID HASKELL to statically guarantee the absence of such exceptions, *i.e.*, to prove various functions *total*.

1.2.1 Specifying Totality

First, let us see how to specify the notion of totality inside LIQUID HASKELL. Consider the source of the above exception:

```
head :: [a] → a
head (x:_) = x
```

Most of the work towards totality checking is done by the translation to GHC's Core, in which every function *is* total, but may explicitly call an *error* function that takes as input a string that describes the source of the pattern-match failure and throws an exception. For example `head` is translated into

```
head d = case d of
           x:xs → x
           []   → patError "head"
```

Since every core function is total, but may explicitly call error functions, to prove that the source function is total, it suffices to prove that `patError` will *never* be called. We can specify this requirement by giving the error functions a `false` pre-condition:

```
patError :: {v:String | False } → a
```

The pre-condition states that the input type is *uninhabited* and so an expression containing a call to `patError` will only type check if the call is *dead code*.

1.2.2 Verifying Totality

The (core) definition of `head` does not typecheck as is; but requires a pre-condition that states that the function is only called with non-empty lists. Formally, we do so by defining the alias

```
predicate NonEmp X = 0 < len X
```

and then stipulating that

```
head :: {v : [a] | NonEmp v} → a
```

To verify the (core) definition of `head`, LIQUID HASKELL uses the above signature to check the body in an environment

```
d :: {0 < len d}
```

When `d` is matched with `[]`, the environment is strengthened with the corresponding refinement from the definition of `len`, *i.e.*,

```
d :: {0 < (len d) ∧ (len d) = 0}
```

Since the formula above is a contradiction, LIQUID HASKELL concludes that the call to `patError` is dead code, and thereby verifies the totality of `head`. Of course, now we have pushed the burden of proof onto clients of `head` – at each such site, LIQUID HASKELL will check that the argument passed in is indeed a `NonEmp` list, and if it successfully does so, then we, at any uses of `head`, can rest assured that `head` will never throw an exception.

Refinements and Totality While the `head` example is quite simple, in general, refinements make it easy to prove totality in complex situations, where we must track dependencies between inputs and outputs. For example, consider the `risers` function from [65]:

```
risers []       = []
risers [x]      = [[x]]
risers (x:y:zs)
  | x ≤ y       = (x:s) : ss
  | otherwise   = [x] : (s:ss)
  where
    s:ss    = risers (y:etc)
```

The pattern match on the last line is partial; its core translation is

```
let (s, ss) = case risers (y:etc) of
                s:ss → (s, ss)
                []   → patError "..."
```

What if `risers` returns an empty list? Indeed, `risers` *does*, on occasion, return an empty list per its first equation. However, on close inspection, it turns out that *if* the input is non-empty, *then* the output is also non-empty. Happily, we can specify this as:

```
risers :: l:_ → {v:_ | NonEmp l ⇒ NonEmp v}
```

LIQUID HASKELL verifies that `risers` meets the above specification, and hence that the `patError` is dead code as at that site, the scrutinee is obtained from calling `risers` with a `NonEmp` list.

Non-Emptiness via Measures Instead of describing non-emptiness indirectly using `len`, a user could a special measure:

```
measure nonEmp  :: [a] → Prop
  nonEmp (x:xs)   = True
  nonEmp []       = False


predicate NonEmp X = nonEmp X
```

After which, verification would proceed analagous to the above.

Total Totality Checking `patError` is one of many possible errors thrown by non-total functions. `Control.Exception.Base` has several others including `recSelError`, `irrefutPatError`, *etc.* which serve the purpose of making core translations total. Rather than hunt down and

specify `False` preconditions one by one, the user may automatically turn on totality checking by invoking LIQUID HASKELL with the `--totality` command line option, at which point the tool systematically checks that all the above functions are indeed dead code, and hence, that all definitions are total.

1.2.3 Case Studies

We verified totality of two libraries: `HsColour` and `Data.Map`, earlier versions of which had previously been proven total by `catch` [65].

`Data.Map` is a widely used library for (immutable) key-value maps, implemented as balanced binary search trees. Totality verification of `Data.Map` was quite straightforward. We had already verified termination and the crucial binary search invariant 3. To verify totality it sufficed to simply re-run verification with the `--totality` argument. All the important specifications were already captured by the types, and no additional changes were needed to prove totality.

This case study illustrates an advantage of LIQUID HASKELL over specialized provers (*e.g.*, `catch` [65]): it can be used to prove totality, termination and functional correctness at the same time, facilitating a nice reuse of specifications for multiple tasks.

`HsColour` is a library for generating syntax-highlighted LATEX and HTML from Haskell source files. Checking `HsColour` was not so easy, as in some cases assumptions are used about the structure of the input data: For example, `ACSS.splitSrcAndAnnos` handles an input list of `String`s and assumes that whenever a specific `String` (say `breakS`) appears then at least two `String`s (call them `mname` and `annots`) follow it in the list. Thus, for a list `ls` that starts with `breakS` the irrefutable pattern `(_:mname:annots) = ls` should be total. Though possible, it is currently it is somewhat cumbersome to specify such properties. As an easy and practical solution, to prove totality, we added a dynamic check that validates that the length of the input `ls` exceeds 2.

In other cases assertions were imposed via monadic checks, *e.g.* `HsColour.hs` reads the input arguments and checks their well-formedness using

```
when (length f > 1) $ errorOut "..."
```

Currently LIQUID HASKELL does not support monadic reasoning that allows assuming that `(length f ≤ 1)` holds when executing the action *following* the `when` check. Finally, code modifications were required to capture properties that are cumbersome to express with LIQUID HASKELL. For example, `trimContext` checks if there is an element that satisfies `p` in the list `xs`; if so it defines `ys = dropWhile (not . p) xs` and computes `tail ys`. By the check we know that `ys` has at least one element, the one that satisfies `p`. Due to the complexity of this property, we preferred to rewrite the specific code in a more verification friendly version.

On the whole, while proving totality can be cumbersome (as in `HsColour`) it is a nice side benefit of refinement type checking and can sometimes be a fully automatic corollary of establishing more interesting safety properties (as in `Data.Map`).

1.3 Termination

Program divergence is, more often than not, a bug rather than a feature. To account for the common cases, by default, LIQUID HASKELL proves termination of each recursive function. Fortunately, refinements make this onerous task quite straightforward. We need simply associate a *well-founded termination metric* on the function's parameters, and then use refinement typing to check that the metric strictly decreases at each recursive call. In practice, due to a careful choice of defaults, this amounts to about a line of termination-related hints per hundred lines of source. In Chapter 2 we prove soundness of our refinement type based termination checker and also we explain how soundness of LIQUID HASKELL crucially depends on the termination checker. Here, we provide an overview on how one can use LIQUID HASKELL to prove termination.

Simple Metrics As a starting example, consider the `fac` function

```
fac :: n:Nat → Nat / [n]
fac 0 = 1
fac n = n * fac (n-1)
```

The termination metric is simply the parameter `n`; as `n` is non-negative and decreases at the recursive call, LIQUID HASKELL verifies that `fac` will terminate. We specify the termination metric in the type signature with the `/[n]`.

Termination checking is performed at the same time as regular type checking, as it can be reduced to refinement type checking with a special terminating fixpoint combinator 2. Thus, if LIQUID HASKELL fails to prove that a given termination metric is well-formed and decreasing, it will report a `Termination Check Error`. At this point, the user can either debug the specification, or mark the function as non-terminating.

Termination Expressions Sometimes, no single parameter decreases across recursive calls, but there is some *expression* that forms the decreasing metric. For example recall `range lo hi` (from § 1.1.2) which returns the list of `Int`s from `lo` to `hi`:

```
range lo hi
  | lo < hi   = lo : range (lo+1) hi
  | otherwise = []
```

Here, neither parameter is decreasing (indeed, the first one is increasing) but `hi-lo` decreases across each call. To account for such cases, we can specify as the termination metric a (refinement logic) expression over the function parameters. Thus, to prove termination, we could type `range` as:

```
lo:Int → hi:Int → [(Btwn lo hi)] / [hi-lo]
```

Lexicographic Termination The Ackermann function

```
ack m n
  | m == 0    = n + 1
  | n == 0    = ack (m-1) 1
  | otherwise = ack (m-1) (ack m (n-1))
```

is curious as there exists no simple, natural-valued, termination metric that decreases at each recursive call. However `ack` terminates because at each call *either* `m` decreases *or* `m` remains the same and `n` decreases. In other words, the pair `(m,n)` strictly decreases according to a *lexicographic* ordering. Thus LIQUID HASKELL supports termination metrics that are a *sequence of* termination expressions. For example, we can type `ack` as:

```
ack :: m:Nat → n:Nat → Nat / [m, n]
```

At each recursive call LIQUID HASKELL uses a lexicographic ordering to check that the sequence of termination expressions is decreasing (and well-founded in each component).

Mutual Recursion The lexicographic mechanism lets us check termination of mutually recursive functions, *e.g.* `isEven` and `isOdd`

```
isEven 0 = True
isEven n = isOdd $ n-1


isOdd n  = not $ isEven n
```

Each call terminates as either `isEven` calls `isOdd` with a decreasing parameter, *or* `isOdd` calls `isEven` with the same parameter, expecting the latter to do the decreasing. For termination, we type:

```
isEven :: n:Nat → Bool / [n, 0]
isOdd  :: n:Nat → Bool / [n, 1]
```

To check termination, LIQUID HASKELL verifies that at each recursive call the metric of the caller is less than the metric of the callee. When `isEven` calls `isOdd`, it proves that the caller's metric, namely `[n,0]` is greater than the callee's `[n-1,1]`. When `isOdd` calls `isEven`, it proves that the caller's metric `[n,1]` is greater than the callee's `[n,0]`, thereby proving the mutual recursion always terminates.

Recursion over Data Types The above strategies generalize easily to functions that recurse over (finite) data structures like arrays, lists, and trees. In these cases, we simply use *measures* to project the structure onto `Nat`, thereby reducing the verification to the previously seen cases. For example, we can prove that `map`

```
map f (x:xs) = f x : map f xs
map f []     = []
```

terminates, by typing `map` as

```
(a → b) → xs:[a] → [b] / [len xs]
```

i.e., by using the measure `len xs`, from § 1.1.3, as the metric.

Generalized Metrics Over Datatypes In many functions there is no single argument whose measure provably decreases. Consider

```
merge (x:xs) (y:ys)
  | x < y     = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys
```

from the homonymous sorting routine. Here, neither parameter decreases, but the *sum* of their sizes does. To prove termination, we can type `merge` as:

```
xs:[a] → ys:[a] → [a] / [len xs + len ys]
```

Putting it all Together The above techniques can be combined to prove termination of the mutually recursive quick-sort (from [105])

```
qsort (x:xs)   = qpart x xs [] []
qsort []       = []

qpart x (y:ys) l r
  | x > y      = qpart x ys (y:l) r
  | otherwise  = qpart x ys l (y:r)
qpart x [] l r = app x (qsort l) (qsort r)

app k []     z = k : z
app k (x:xs) z = x : app k xs z
```

`qsort (x:xs)` calls `qpart x xs` to partition `xs` into two lists `l` and `r` that have elements less and greater or equal than the pivot `x`, respectively. When `qpart` finishes partitioning it mutually recursively calls `qsort` to sort the two list and appends the results with `app`. LIQUID HASKELL proves sortedness as well [98] but let us focus here on termination. To this end, we type the functions as:

```
qsort :: xs:_ → _
      / [len xs, 0]

qpart :: _ → ys:_ → l:_ → r:_ → _
      / [len ys + len l + len r, 1 + len ys]
```

As before, LIQUID HASKELL checks that at each recursive call the caller's metric is less than the callee's. When `qsort` calls `qpart` the length of the unsorted list `len (x:xs)` exceeds the `len xs + len [] + len []`. When `qpart` recursively calls itself the first component of the metric is the same, but the length of the unpartitioned list decreases, *i.e.* `1 + len y:ys` exceeds `1 + len ys`. Finally, when `qpart` calls `qsort` we have `len ys + len l + len r` exceeds both `len l` and `len r`, thereby ensuring termination.

Automation: Default Size Measures The `qsort` example illustrates that while LIQUID HASKELL is very expressive, devising appropriate termination metrics can be tricky. Fortunately, such patterns are very uncommon, and the vast majority of cases in real world programs are just structural recursion on a datatype. LIQUID HASKELL automates termination proofs for this common case, by allowing users to specify a *default size measure* for each data type, *e.g.* `len` for `[a]`. Now, if no explicit termination metric is given, by default LIQUID HASKELL assumes that the *first* argument whose type has an associated size measure decreases. Thus, in the above, we need not specify metrics for `fac` or `map` as the size measure is automatically used to prove termination. This heuristic suffices to *automatically* prove 67% of recursive functions terminating.

Disabling Termination Checking In `Haskell`'s lazy setting not all functions are terminating. LIQUID HASKELL provides two mechanisms the disable termination proving. A user can disable checking a single function by marking that function as lazy. For example, specifying `lazy repeat` tells the tool to not prove `repeat` terminates. Optionally, a user can disable termination checking for a whole module by using the command line argument `--no-termination` for the entire file.

1.4 Memory Safety

The terms "Haskell" and "pointer arithmetic" rarely occur in the same sentence, yet many Haskell programs are constantly manipulating pointers under the hood by way of using the `Bytestring` and `Text` libraries. These libraries sacrifice safety for (much needed) speed and are natural candidates for verification through LIQUID HASKELL.

1.4.1 Bytestring

The single most important aspect of the `Bytestring` library, our first case study, is its pervasive intermingling of high level abstractions like higher-order loops, folds, and fusion, with low-level pointer manipulations in order to achieve high-performance. `Bytestring` is an appealing target for evaluating LIQUID HASKELL, as refinement types are an ideal way to statically ensure the correctness of the delicate pointer manipulations, errors in which lie below the scope of dynamic protection.

The library spans 8 files (modules) totaling about 3,500 lines. We used LIQUID HASKELL to verify the library by giving precise types describing the sizes of internal pointers and bytestrings. These types are used in a modular fashion to verify the implementation of functional correctness properties of higher-level API functions which are built using lower-level internal operations. Next, we show the key invariants and how LIQUID HASKELL reasons precisely about pointer arithmetic and higher-order codes.

Key Invariants A (strict) `ByteString` is a triple of a `payload` pointer, an `offset` into the memory buffer referred to by the pointer (at which the string actually "begins") and a `length` corresponding to the number of bytes in the string, which is the size of the buffer *after* the `offset`, that corresponds to the string. We define a measure for the *size* of a `ForeignPtr`'s buffer, and use it to define the key invariants as a refined datatype

```
measure fplen  :: ForeignPtr a → Int
data ByteString = PS
   { pay :: ForeignPtr Word8
   , off :: {v:Nat | v       ≤ fplen pay }
   , len :: {v:Nat | off + v ≤ fplen pay } }
```

The definition states that the offset is a `Nat` no bigger than the size of the `payload`'s buffer, and that the sum of the `offset` and non-negative `length` is no more than the size of the payload buffer. Finally, we encode a `ByteString`'s size as a measure.

```
measure bLen   :: ByteString → Int
bLen (PS p o l) = l
```

Specifications We define a type alias for a `ByteString` whose length is the same as that of another, and use the alias to type the API function `copy`, which clones `ByteStrings`.

```
type ByteStringEq B = {v:ByteString | (bLen v) = (bLen B)}


copy :: b:ByteString → ByteStringEq b
copy (PS fp off len)
  = unsafeCreate len $ \p →
      withForeignPtr fp $ \f →
        memcpy len p (f `plusPtr` off)
```

Pointer Arithmetic The simple body of `copy` abstracts a fair bit of internal work. `memcpy sz dst src`, implemented in C and accessed via the FFI is a potentially dangerous, low-level operation, that copies `sz` bytes starting *from* an address `src` *into* an address `dst`. Crucially, for safety, the regions referred to be `src` and `dst` must be larger than `sz`. We capture this requirement by defining a type alias `PtrN a N` denoting GHC pointers that refer to a region bigger than `N` bytes, and then specifying that the destination and source buffers for `memcpy` are large enough.

```
type PtrN a N = {v:Ptr a | N ≤ (plen v)}
memcpy :: sz:CSize → dst:PtrN a siz
                   → src:PtrN a siz
                   → IO ()
```

The actual output for `copy` is created using the internal function `unsafeCreate` which is a wrapper around.

```
create :: l:Nat → f:(PtrN Word8 l → IO ())
       → IO (ByteStringN l)
create l f = do
    fp <- mallocByteString l
    withForeignPtr fp $ \p → f p
    return $! PS fp 0 l
```

The type of `f` specifies that the action will only be invoked on a pointer of length at least `l`, which is verified by propagating the types of `mallocByteString` and `withForeignPtr`. The fact that the action is only invoked on such pointers is used to ensure that the value `p` in the body

of `copy` is of size `l`. This, and the `ByteString` invariant that the size of the payload `fp` exceeds the sum of `off` and `len`, ensures safety of the `memcpy` call.

Interfacing with the Real World The above illustrates how LIQUID HASKELL analyzes code that interfaces with the "real world" via the C FFI. We specify the behavior of the world via a refinement typed interface. These types are then assumed to hold for the corresponding functions, *i.e.* generate pre-condition checks and post-condition guarantees at usage sites within the Haskell code.

Higher Order Loops `mapAccumR` combines a `map` and a `foldr` over a `ByteString`. The function uses non-trivial recursion, and demonstrates the utility of abstract-interpretation based inference.

```
mapAccumR f z b = unSP $ loopDown (mapAccumEFL f) z b
```

To enable fusion [23] `loopDown` uses a higher order `loopWrapper` to iterate over the buffer with a `doDownLoop` action:

```
doDownLoop f acc0 src dest len = loop (len-1) (len-1) acc0
  where
   loop :: s:_ → _ → _ → _ / [s+1]
   loop s d acc
     | s < 0
     = return (acc :*: d+1 :*: len - (d+1))
     | otherwise
     = do x <- peekByteOff src s
          case f acc x of
            (acc′ :*: NothingS) →
                 loop (s-1) d acc′
            (acc′ :*: JustS x′) →
                 pokeByteOff dest d x′
              >> loop (s-1) (d-1) acc′
```

The above function iterates across the `src` and `dst` pointers from the right (by repeatedly decrementing the offsets `s` and `d` starting at the high `len` down to `-1`). Low-level reads and writes are carried out using the potentially dangerous `peekByteOff` and `pokeByteOff` respectively. To ensure safety, we type these low level operations with refinements stating that they are only

invoked with valid offsets `VO` into the input buffer `p`.

```
type VO P    = {v:Nat | v < plen P}
peekByteOff :: p:Ptr b → VO p → IO a
pokeByteOff :: p:Ptr b → VO p → a → IO ()
```

The function `doDownLoop` is an internal function. LIQUID HASKELL, via abstract interpretation [79], infers that (1) `len` is less than the sizes of `src` and `dest`, (2) `f` (here, `mapAccumEFL`) always returns a `JustS`, so (3) both the source and the destination offsets satisfy $0 \leq \texttt{s},\texttt{d} < \texttt{len}$, (4) the generated `IO` action returns a triple `(acc :*: 0 :*: len)`, thereby proving the safety of the accesses in `loop` *and* verifying that `loopDown` and the API function `mapAccumR` return a `Bytestring` whose size equals its input's.

To prove *termination*, we add a *termination expression* `s+1` which is always non-negative and decreases at each call.

Nested Data `group` splits a string like `"aart"` into the list `["aa","r","t"]`, *i.e.* a list of (a) non-empty `ByteStrings` whose (b) total length equals that of the input. To specify these requirements, we define a measure for the total length of strings in a list and use it to define the list of *non-empty* strings whose total length equals that of another string:

```
measure bLens :: [ByteString] → Int
  bLens ([])     = 0
  bLens (x:xs)   = bLen x + bLens xs

type ByteStringNE    = {v:ByteString | bLen v > 0}
type ByteStringsEq B = {v:[ByteStringNE] | bLens v = bLen b}
```

LIQUID HASKELL uses the above to verify that

```
group :: b:ByteString → ByteStringsEq b
group xs
 | null xs   = []
 | otherwise = let x        = unsafeHead xs
                   xs'      = unsafeTail xs
                   (ys, zs) = spanByte x xs'
               in (y `cons` ys) : group zs
```

The example illustrates why refinements are critical for proving termination. LIQUID HASKELL determines that `unsafeTail` returns a *smaller* `ByteString` than its input and that each element returned by `spanByte` is no bigger than the input, concluding that `zs` is smaller than `xs`, hence checking the body under the termination-weakened environment.

To justify the output type, let's look at `spanByte`, which splits strings into a pair:

```
spanByte c ps@(PS x s l)
  = inlinePerformIO $ withForeignPtr x $
        \p → go (p `plusPtr` s) 0
  where
    go :: _ → i:_ → _ / [l-i]
    go p i
      | i ≥ l    = return (ps, empty)
      | otherwise = do
          c′ <- peekByteOff p i
          if c /= c′
            then let b1 = unsafeTake i ps
                     b2 = unsafeDrop i ps
                 in  return (b1, b2)
            else go p (i+1)
```

Via inference, LIQUID HASKELL verifies the safety of the pointer accesses, and determines that the sum of the lengths of the output pair of `ByteStrings` equals that of the input `ps`. `go` terminates as `l-i` is a well-founded decreasing metric.

1.4.2 Text

Next we present a brief overview of the verification of `Text`, which is the standard library used for serious unicode text processing. `Text` uses byte arrays and stream fusion to guarantee performance while providing a high-level API. In our evaluation of LIQUID HASKELL on `Text`,we focused on two types of properties: (1) the safety of array index and write operations, and (2) the functional correctness of the top-level API. These are both made more interesting by the fact that `Text` internally encodes characters using UTF-16, in which characters are stored in either two or four bytes. `Text` is a vast library spanning 39 modules and 5,700 lines of code,

however we focus on the 17 modules that are relevant to the above properties. While we have verified exact functional correctness size properties for the top-level API, we focus here on the low-level functions and interaction with unicode.

Arrays and Texts A `Text` consists of an (immutable) `Array` of 16-bit words, an offset into the `Array`, and a length describing the number of `Word16`s in the `Text`. The `Array` is created and filled using a *mutable* `MArray`. All write operations in `Text` are performed on `MArray`s in the `ST` monad, but they are *frozen* into `Array`s before being used by the `Text` constructor. We write a measure for the size of an `MArray` and use it to type the write and freeze operations.

```
measure malen       :: MArray s → Int
predicate EqLen A MA = alen A = malen MA
predicate Ok I A     = 0 ≤ I < malen A
type VO A            = {v:Int| Ok v A}

unsafeWrite  :: m:MArray s
             → VO m → Word16 → ST s ()
unsafeFreeze :: m:MArray s
             → ST s {v:Array | EqLen v m}
```

Reasoning about Unicode The function `writeChar` (abbreviating the function `unsafeWrite` from `UnsafeChar`) writes a `Char` into an `MArray`. `Text` uses UTF-16 to represent characters internally, meaning that every `Char` will be encoded using two or four bytes (one or two `Word16`s).

```
writeChar marr i c
    | n < 0x10000 = do
        unsafeWrite marr i (fromIntegral n)
        return 1
    | otherwise = do
        unsafeWrite marr i lo
        unsafeWrite marr (i+1) hi
        return 2
    where n = ord c
          m = n - 0x10000
          lo = fromIntegral
```

```
            $ (m `shiftR` 10) + 0xD800
        hi = fromIntegral
            $ (m .&. 0x3FF) + 0xDC00
```

The UTF-16 encoding complicates the specification of the function as we cannot simply require `i` to be less than the length of `marr`; if `i` were `malen marr - 1` and `c` required two `Word16`s, we would perform an out-of-bounds write. We account for this subtlety with a predicate that states there is enough `Room` to encode `c`.

```
predicate OkN I A N  = Ok (I+N-1) A
predicate Room I A C = if ord C < 0x10000
                       then OkN I A 1
                       else OkN I A 2

type OkSiz I A = {v:Nat  | OkN  I A v}
type OkChr I A = {v:Char | Room I A v}
```

`Room i marr c` says "if `c` is encoded using one `Word16`, then `i` must be less than `malen marr`, otherwise `i` must be less than `malen marr - 1`." `OkSiz I A` is an alias for a valid number of `Word16`s remaining after the index `I` of array `A`. `OkChr` specifies the `Char`s for which there is room (to write) at index `I` in array `A`. The specification for `writeChar` states that given an array `marr`, an index `i`, and a valid `Char` for which there is room at index `i`, the output is a monadic action returning the number of `Word16` occupied by the `char`.

```
writeChar :: marr:MArray s
          → i:Nat
          → OkChr i marr
          → ST s (OkSiz i marr)
```

Bug Thus, clients of `writeChar` should only call it with suitable indices and characters. Using LIQUID HASKELL we found an error in one client, `mapAccumL`, which combines a map and a fold over a `Stream`, and stores the result of the map in a `Text`. Consider the inner loop of `mapAccumL`.

```
outer arr top = loop
 where
  loop !z !s !i =
```

```
case next0 s of
  Done          → return (arr, (z,i))
  Skip s'       → loop z s' i
  Yield x s'
    | j ≥ top   → do
      let top' = (top + 1) `shiftL` 1
      arr' <- new top'
      copyM arr' 0 arr 0 top
      outer arr' top' z s i
    | otherwise → do
      let (z',c) = f z x
      d <- writeChar arr i c
      loop z' s' (i+d)
    where j | ord x < 0x10000 = i
            | otherwise       = i + 1
```

Let's focus on the `Yield x s'` case. We first compute the maximum index `j` to which we will write and determine the safety of a write. If it is safe to write to `j` we call the provided function `f` on the accumulator `z` and the character `x`, and write the *resulting* character `c` into the array. However, we know nothing about `c`, in particular, whether `c` will be stored as one or two `Word16`s! Thus, LIQUID HASKELL flags the call to `writeChar` as *unsafe*. The error can be fixed by lifting `f z x` into the `where` clause and defining the write index `j` by comparing `ord c` (not `ord x`). LIQUID HASKELL (and the authors) readily accepted our fix.

1.5 Functional Correctness Invariants

So far, we have considered a variety of general, application independent correctness criteria. Next, let us see how we can use LIQUID HASKELL to specify and statically verify critical, application specific correctness properties, using two illustrative case studies: red-black trees and the stack-set data structure introduced in the `xmonad` system.

1.5.1 Red-Black Trees

Red-Black trees have several non-trivial invariants that are ideal for illustrating the effectiveness of refinement types and contrasting with existing approaches based on GADTs [45]. The structure can be defined via the following Haskell type:

```
data Col    = R | B
data Tree a = Leaf
            | Node Col a (Tree a) (Tree a)
```

However, a `Tree` is a valid Red-Black tree only if it satisfies three crucial invariants:

- ***Order:*** The keys must be binary-search ordered, *i.e.* the key at each node must lie between the keys of the left and right subtrees of the node,
- ***Color:*** The children of every *red* `Node` must be colored *black*, where each `Leaf` can be viewed as black,
- ***Height:*** The number of black nodes along any path from each `Node` to its `Leaf`s must be the same.

Red-Black trees are especially tricky as various operations create trees that can temporarily violate the invariants. Thus, while the above invariants can be specified with singletons and GADTs, encoding all the properties (and the temporary violations) results in a proliferation of data constructors that can somewhat obfuscate correctness. In contrast, with refinements, we can specify and verify the invariants in isolation (if we wish) and can trivially compose them simply by *conjoining* the refinements.

Color Invariant To specify the color invariant, we define a *black-rooted tree* as:

```
measure isB           :: Tree a → Prop
  isB (Node c x l r)  = c == B
  isB (Leaf)          = True
```

and then we can describe the color invariant simply as:

```
measure isRB          :: Tree a → Prop
  isRB (Leaf)         = True
```

```
isRB (Node c x l r) = isRB l ∧ isRB r ∧
                      c = R ⇒ (isB l ∧ isB r)
```

The insertion and deletion procedures create intermediate *almost* red-black trees where the color invariant may be violated at the root. Rather than create new data constructors we define almost red-black trees with a measure that just drops the invariant at the root:

```
measure almostRB          :: Tree a → Prop
  almostRB (Leaf)         = True
  almostRB (Node c x l r) = isRB l ∧ isRB r
```

Height Invariant To specify the height invariant, we define a black-height measure:

```
measure bh          :: Tree a → Int
  bh (Leaf)         = 0
  bh (Node c x l r) = bh l + if c = R then 0 else 1
```

and we can now specify black-height balance as:

```
measure isBal          :: Tree a → Prop
  isBal (Leaf)         = true
  isBal (Node c x l r) = bh l = bh r
                       ∧ isBH l ∧ isBH r
```

Note that `bh` only considers the left sub-tree, but this is legitimate, because `isBal` will ensure the right subtree has the same `bh`.

Order Invariant We refine the data definition of `Tree` to encode the ordering property:

```
data Tree a
  = Leaf
  | Node { c   :: Col
         , key :: a
         , lt  :: Tree {v:a | v < key }
         , rt  :: Tree {v:a | key < v } }
```

Composing Invariants Finally, we can compose the invariants and define a Red-Black tree with the alias:

```
type RBT a = {v:Tree a | isRB v ∧ isBal v}
```

An almost Red-Black tree is the above with `isRB` replaced with `almostRB`, *i.e.* does not require any new types or constructors. If desired, we can ignore a particular invariant simply by replacing the corresponding refinement above with `true`. Given the above – and suitable signatures LIQUID HASKELL verifies the various insertion, deletion and rebalancing procedures for a Red-Black Tree library.

1.5.2 Stack Sets in XMonad

`xmonad` is a dynamically tiling `X11` window manager that is written and configured in Haskell. The set of windows managed by XMonad is organized into a hierarchy of types. At the lowest level we have a *set* of windows `a` represented as a `Stack a`

```
data Stack a = Stack { focus :: a
                     , up    :: [a]
                     , down  :: [a] }
```

The above is a zipper [40] where `focus` is the "current" window and `up` and `down` the windows "before" and "after" it. Each `Stack` is wrapped inside a `Workspace` that also has information about layout and naming:

```
data Workspace i l a = Workspace
   { tag    :: i
   , layout :: l
   , stack  :: Maybe (Stack a) }
```

which is in turn, wrapped inside a `Screen`:

```
data Screen i l a sid sd = Screen
  { workspace    :: Workspace i l a
  , screen       :: sid
  , screenDηil :: sd }
```

The set of all screens is represented by the top-level zipper:

```
data StackSet i l a sid sd = StackSet
  { cur :: Screen i l a sid sd
  , vis :: [Screen i l a sid sd]
  , hid :: [Workspace i l a]
```

```
  , flt :: M.Map a RationalRect }
```

Key Invariant: Uniqueness of Windows The key invariant for the `StackSet` type is that each window `a` should appear at most once in a `StackSet i l a sid sd`. That is, a window should *not be duplicated* across stacks or workspaces. Informally, we specify this invariant by defining a measure for the *set of elements* in a list, `Stack`, `Workspace` and `Screen`, and then we use that measure to assert that the relevant sets are disjoint.

Specification: Unique Lists To specify that the set of elements in a list is unique, *i.e.* there are no duplicates in the list we first define a measure denoting the set using Z3's [24] built-in theory of sets:

```
measure elts  :: [a] → Set a
  elts ([])   = emp
  elts (x:xs) = cup (sng x) (elts xs)
```

Now, we can use the above to define uniqueness:

```
measure isUniq  :: [a] → Prop
  isUniq ([])   =  true
  isUniq (x:xs) =  notIn x xs ∧ isUniq xs
```

where `notIn` is an abbreviation:

```
predicate notIn X S = not (mem X (elts S))
```

Specification: Unique Stacks We can use `isUniq` to define unique, *i.e.*, duplicate free, `Stacks` as:

```
data Stack a = Stack
 { focus :: a
 , up    :: {v:[a] | Uniq1 v focus}
 , down  :: {v:[a] | Uniq2 v focus up} }
```

using the aliases

```
predicate Uniq1 V X    = isUniq V  ∧ notIn X V
predicate Uniq2 V X Y  = Uniq1 V X ∧ disjoint Y V
predicate disjoint X Y = cap (elts X) (elts Y) = emp
```

i.e. the field up is a unique list of elements different from `focus`, and the field `down` is additionally disjoint from up.

Specification: Unique StackSets It is straightforward to lift the `elts` measure to the `Stack` and the wrapper types `Workspace` and `Screen`, and then correspondingly lift `isUniq` to `[Screen]` and `[Workspace]`. Having done so, we can use those measures to refine the type of `StackSet` to stipulate that there are no duplicates:

```
type UniqStackSet i l a sid sd
  = {v: StackSet i l a sid sd | NoDups v}
```

using the predicate aliases

```
predicate NoDups V
  =  disjoint3 (hid V) (cur V) (vis V)
  ∧ isUniq (vis V) ∧ isUniq (hid V)


predicate disjoint3 X Y Z
  =  disjoint X Y ∧ disjoint Y Z ∧ disjoint X Z
```

LIQUID HASKELL automatically turns the record selectors of refined data types to measures that return the values of appropriate fields, hence `hid x` (resp. `cur x`, `vis x`) are the values of the `hid`, `cur` and `vis` fields of a `StackSet` named `x`.

Verification LIQUID HASKELL uses the above refined type to verify the key invariant, namely, that no window is duplicated. Three key actions of the, eventually successful, verification process can be summarized as follows:

- *Strengthening library functions.* `xmonad` repeatedly concatenates the lists of a Stack. To prove that for some `s:Stack a`, `(up s ++ down s)` is a unique list, the type of `(++)` needs to capture that concatenation of two unique and disjoint lists is a unique list. For verification, we assumed that Prelude's `(++)` satisfies this property. But, not all arguments of `(++)` are unique disjoint lists: `"StackSet" ++ "error"` is a trivial example that does not satisfy the assumed preconditions of `(++)` thus creating a type error. Currently, LIQUID HASKELL does not support intersection types, thus we used an unrefined `(++.)` variant of `(++)` for such cases.

- *Restrict the functions' domain.* `modify` is a `maybe`-like function that given a default value `x`, a function `f`, and a StackSet `s`, applies `f` on the `Maybe` values inside `s`.

```
modify :: x:{v:Maybe (Stack a) | isNothing v}
       → (y:Stack a → Maybe {v:Stack a | SubElts v y})
       → UniqStackSet i l a s sd
       → UniqStackSet i l a s sd
```

  Since inside the StackSet `s` each `y:Stack a` could be replaced with either the default value `x` or with `f y`, we need to ensure that both these alternatives will not insert duplicates. This imposes the curious precondition that the default value should be `Nothing`.

- *Code inlining* Given a tag `i` and a StackSet `s`, `view i s` will set the current Screen to the screen with tag `i`, if such a screen exists in `s`. Below is the original definition for `view` in case when a screen with tag `i` exists in visible screens

```
view :: (Eq s, Eq i) ⇒ i
     → StackSet i l a s sd
     → StackSet i l a s sd
view i s
  | Just x <- find ((i==).tag.workspace) (visible s)
  = s { current = x
      , visible = current s
                : deleteBy (equating screen) x (visible s) }
```

  Verification of this code is difficult as we cannot suitably type `find`. Instead we *inline* the call to `find` and the field update into a single recursive function `raiseIfVisible i s` that in-place replaces `x` with the current screen.

Finally, `xmonad` comes with an extensive suite of QuickCheck properties, that were formally verified in Coq [89]. In future work 8, it would be interesting to do a similar verification with LIQUID HASKELL, to compare the refinement types to proof-assistants.

1.6 Evaluation

We now present a quantitative evaluation of LIQUID HASKELL.

Table 1.1. A quantitative evaluation of our experiments. **Version** is version of the checked library. **LOC** is the number of non-comment lines of source code as reported by `sloccount`. **Mod** is the number of modules in the benchmark and **Fun** is the number of functions. **Specs** is the number (/ line-count) of type specifications and aliases, data declarations, and measures provided. **Annot** is the number (/ line-count) of other annotations provided, these include invariants and hints for the termination checker. **Qualif** is the number (/ line-count) of provided qualifiers. **Time (s)** is the time, in seconds, required to run LIQUID HASKELL.

Module	Version	LOC	Mod	Fun	Specs	Annot	Qualif	Time (s)
GHC.LIST	7.4.1	309	1	66	29 / 38	6 / 6	0 / 0	15
DATA.LIST	4.5.1.0	504	1	97	15 / 26	6 / 6	3 / 3	11
DATA.MAP.BASE	0.5.0.0	1396	1	180	125 / 173	13 / 13	0 / 0	174
DATA.SET.SPLAY	0.1.1	149	1	35	27 / 37	5 / 5	0 / 0	27
HSCOLOUR	1.20.0.0	1047	16	234	19 / 40	5 / 5	1 / 1	196
XMONAD.STACKSET	0.11	256	1	106	74 / 213	3 / 3	4 / 4	27
BYTESTRING	0.9.2.1	3505	8	569	307 / 465	55 / 55	47 / 124	294
TEXT	0.11.2.3	3128	17	493	305 / 717	52 / 54	49 / 97	499
VECTOR-ALGORITHMS	0.5.4.2	1218	10	99	76 / 266	9 / 9	13 / 13	89
Total		11512	56	1879	977 / 1975	154 / 156	117 / 242	1336

1.6.1 Results

We have used the following libraries as benchmarks:

- `GHC.List` and `Data.List`, which together implement many standard list operations; we verify various size related properties,
- `Data.Set.Splay`, which implements a splay-tree based functional set data type; we verify that all interface functions terminate and return well ordered trees,
- `Data.Map.Base`, which implements a functional map data type; we verify that all interface functions terminate and return binary-search ordered trees [98],
- `HsColour`, a syntax highlighting program for Haskell code, we verify totality of all functions (§ 1.2),
- `XMonad`, a tiling window manager for X11, we verify the uniqueness invariant of the core datatype, as well as some of the QuickCheck properties (§ 1.5.2),
- `Bytestring`, a library for manipulating byte arrays, we verify termination, low-level memory safety, and high-level functional correctness properties (§ 1.4.1),

- `Text`, a library for high-performance unicode text processing; we verify various pointer safety and functional correctness properties (§ 1.4.2), during which we find a subtle bug,
- `Vector-Algorithms`, which includes a suite of "imperative" (*i.e.* monadic) array-based sorting algorithms; we verify the correctness of vector accessing, indexing, and slicing *etc.*.

Table 1.1 summarizes our experiments, which covered 56 modules totaling 11,512 non-comment lines of source code and 1,975 lines of specifications. The results are on a machine with an Intel Xeon X5660 and 32GB of RAM (no benchmark required more than 1GB.) The upshot is that LIQUID HASKELL is very effective on real-world code bases. The total overhead due to hints, *i.e.* the sum of **Annot** and **Qualif**, is 3.5% of **LOC**. The specifications themselves are machine checkable versions of the comments placed around functions describing safe usage and behavior, and required around two lines to express on average. While there is much room for improving the running times, the tool is fast enough to be used interactively, verify a handful of API functions and associated helpers in isolation.

1.6.2 Limitations

Our case studies also highlighted several limitations of LIQUID HASKELL. In most cases, we could alter the code slightly to facilitate verification.

Ghost parameters are sometimes needed in order to materialize values that are not needed for the computation, but are necessary to prove various specifications. For example, the `piv` parameter in the `append` function for red-black trees (§ 1.5.1). Bounded Refinement Types (Chapter 4) provide a complete, but unfortunately not elegant way to eliminated ghost parameters.

Fixed-width integer and floating-point numbers LIQUID HASKELL uses the theories of linear arithmetic and real numbers to reason about numeric operations. In some cases this causes us to lose precision, *e.g.* when we have to approximate the behavior of bitwise operations. We could address this shortcoming by using the theory of bit-vectors to model fixed-width integers, but we are unsure of the effect this would have on LIQUID HASKELL's performance.

Functions as Data Several libraries like `Text` encode data structures like (finite) streams using functions, in order to facilitate fusion. Currently, it is not possible to describe sizes of these

structures using measures, as this requires describing the sizes of input-output chains starting at a given seed input for the function. In future work, it will be interesting to extend the measure mechanism to support multiple parameters (*e.g.* a stream and a seed) in order to reason about such structures.

Lazy binders sometimes get in the way of verification. A common pattern in Haskell code is to define *all* local variables in a single `where` clause and use them only in a subset of all branches. LIQUID HASKELL flags a few such definitions as *unsafe*, not realizing that the values will only be demanded in a specific branch. Currently, we manually transform the code by pushing binders inwards to the usage site. This transformation could be easily automated.

Assumes which can be thought of as "hybrid" run-time checks, had to be placed in a couple of cases where the verifier loses information. One source is the introduction of assumptions about mathematical operators that are currently conservatively modeled in the refinement logic (*e.g.* that multiplication is commutative and associative). These may be removed by using more advanced non-linear arithmetic decision procedures.

Error messages are a crucial part of any type-checker. Currently, we report error locations in the provided source file and output the failed constraint(s). In the future errors should be reported using an interactive interface with features including code and type completion and counter example drive error explanation.

Acknowledgments The material of this chapter are adapted from the following publication: N. Vazou, E. Seidel, and R. Jhala, "LiquidHaskell: Experience with Refinement Types in the Real World", Haskell, 2014.

Chapter 2

Soundness Under Lazy Evaluation

Laziness may appear attractive, but work gives satisfaction.

– Anne Frank

When we started the LIQUID HASKELL project, we built on the theory of standard refinement types as implemented for example for ML [106, 7, 79]. Standard refinement types were developed for the eager, *call-by-value* languages, but we presumed that the order of evaluation would surely prove irrelevant and that the soundness guarantees would translate to Haskell's lazy, *call-by-need* regime. We were wrong.

- We start this chapter by showing that standard refinement systems crucially rely on a property of eager languages: when analyzing any term, one can assume that *all* the free variables appearing in the term are bound to *values*. This property lets us check each term in an environment where the free variables are logically constrained according to their refinements. Unfortunately, this property does not hold for lazy evaluation, where free variables can be lazily substituted with arbitrary (potentially diverging) expressions, which breaks soundness (§2.1).

  The two natural paths towards soundness are blocked by challenging problems. The first path is to *conservatively ignore* free variables except those that are guaranteed to be values *e.g.* by pattern matching, `seq` or strictness annotations. While sound, this leads to a drastic loss of precision. The second path is to *explicitly* reason about divergence within the

refinement logic. This would be sound and precise – however it is far from obvious to us how to re-use and extend existing SMT machinery for this purpose. (§2.6)

- Next, we present a novel approach that enables sound and precise checking with existing SMT solvers, using a *stratified* type system that labels binders as potentially diverging or not (§2.3). While previous stratified systems [20] would suffice for soundness, we show how to recover precision by using refinement types to develop a notion of *terminating fixpoint* combinators that allows the type system to automatically verify that a wide variety of recursive functions actually terminate (§2.4).

- Finally, we provide an extensive empirical evaluation of our approach on more than $10,000$ lines of widely used complex Haskell libraries. We have implemented our approach in LIQUID HASKELL, and use it to prove termination of libraries verified in Chapter 1. LIQUID HASKELL is able to prove 96% of all recursive functions terminating, requiring a modest 1.7 lines of termination annotations per 100 lines of code, thereby enabling the sound, precise, and automated verification of functional correctness properties of real-world Haskell code (§2.5).

2.1 Overview

We start with an informal overview of a sound refinement type system for Haskell. After recapitulating the basics of refinement types we illustrate why the classical approach based on verification conditions (VCs) is unsound due to lazy evaluation. Next, we step back to understand precisely how the VCs arise from refinement subtyping and how subtyping is different under eager and lazy evaluation. In particular, we demonstrate that under lazy, but *not* eager, evaluation, the refinement type system, and hence the VCs, must account for divergence. Consequently, we develop a type system that accounts for divergence in a modular and syntactic fashion and illustrate its use via several small examples. Finally, we show how a refinement-based termination analysis is used to improve precision, yielding a highly effective SMT-based verifier for Haskell.

$$
\begin{array}{rrcl}
\textbf{\textit{Refinements}} & r & ::= & \ldots \text{varies} \ldots \\
\textbf{\textit{Basic Types}} & b & ::= & \{v{:}\texttt{Int} \mid r\} \mid \ldots \\
\textbf{\textit{Types}} & \tau & ::= & b \mid x{:}\tau \to \tau \\
\textbf{\textit{Environment}} & \Gamma & ::= & \emptyset \mid x{:}\tau, \Gamma \\
\textbf{\textit{Subtyping}} & & & \Gamma \vdash \tau_1 \preceq \tau_2 \\
\textbf{\textit{Abbreviations}} & & & \\
x{:}\{r\} & & \doteq & x{:}\{x{:}\texttt{Int} \mid r\} \\
\{x \mid r\} & & \doteq & \{x{:}\texttt{Int} \mid r\} \\
\{r\} & & \doteq & \{v{:}\texttt{Int} \mid r\} \\
\{x{:}\{y{:}\texttt{Int} \mid r_y\} \mid r_x\} & & \doteq & \{x{:}\texttt{Int} \mid r_x \wedge r_y[x/y]\} \\
\textbf{\textit{Translation}} & & & \\
(\!|\Gamma \vdash b_1 \preceq b_2|\!) & & \doteq & (\!|\Gamma|\!) \Rightarrow (\!|b_1|\!) \Rightarrow (\!|b_2|\!) \\
(\!|\{x{:}\texttt{Int} \mid r\}|\!) & & \doteq & r \\
(\!|x{:}\{v{:}\texttt{Int} \mid r\}|\!) & & \doteq & \text{“}x \text{ is a value”} \Rightarrow r[x/v] \\
(\!|x{:}(y{:}\tau_y \to \tau)|\!) & & \doteq & \texttt{True} \\
(\!|x_1{:}\tau_1, \ldots, x_n{:}\tau_n|\!) & & \doteq & (\!|x_1{:}\tau_1|\!) \wedge \ldots \wedge (\!|x_n{:}\tau_n|\!)
\end{array}
$$

Figure 2.1. Informal Notation: Types, Subtyping, and VCs.

2.1.1 Standard Refinement Types: From Subtyping to VC

First, let us see how standard refinement type systems [79, 51] will use the aforementioned refinement type aliases `Pos` and `Nat` and the specification for `div` to *accept* `good` and *reject* `bad`. We use the syntax of Figure 2.1, where r is a *refinement expression*, or just *refinement* for short. We will vary the expressiveness of the language of refinements in different parts of this document.

```
good     :: Nat → Nat → Int
good x y = let z = y + 1 in x `div` z


bad      :: Nat → Nat → Int
bad x y  = x `div` y
```

Refinement Subtyping To analyze the body of `bad`, the refinement type system will check that the second parameter `y` has type `Pos` at the call to `div`; formally, that the actual parameter `y` is a

subtype of the type of `div`'s second input, via a subtyping query:

$$\texttt{x}{:}\{\texttt{x}{:}\texttt{Int} \mid \texttt{x} \geq 0\},\ \texttt{y}{:}\{\texttt{y}{:}\texttt{Int} \mid \texttt{y} \geq 0\} \vdash \{\texttt{y}{:}\texttt{Int} \mid \texttt{y} \geq 0\} \preceq \{\texttt{v}{:}\texttt{Int} \mid \texttt{v} > 0\}$$

We use the Abbreviations of Figure 2.1 to simplify the syntax of the queries. So the above query simplifies to:

$$\texttt{x}{:}\{\texttt{x} \geq 0\},\ \texttt{y}{:}\{\texttt{y} \geq 0\} \vdash \{\texttt{v} \geq 0\} \preceq \{\texttt{v} > 0\}$$

Verification Conditions To discharge the above subtyping query, a refinement type system generates a *verification condition* (VC), a logical formula that stipulates that under the assumptions corresponding to the environment bindings, the refinement in the sub-type *implies* the refinement in the super-type. We use the translation $(\!|\cdot|\!)$ shown in Figure 2.1 to reduce a subtyping query to a verification condition. The translation of a basic type into logic is the refinement of the type. The translation of an environment is the conjunction of its bindings. Finally, the translation of a binding $x{:}\tau$ is the embedding of τ guarded by a predicate denoting that "x is a value". For now, let us ignore this guard and see how the subtyping query for `bad` reduces to the *classical* VC:

$$(\texttt{x} \geq 0) \wedge (\texttt{y} \geq 0) \ \Rightarrow\ (\texttt{v} \geq 0) \Rightarrow (\texttt{v} > 0)$$

Refinement type systems are carefully engineered (§2.3) so that (*unlike* with full dependent types) the logic of refinements *precludes* arbitrary functions and only includes formulas from efficiently decidable logics, *e.g.* the quantifier-free logic of linear arithmetic and uninterpreted functions (QF-EUFLIA). Thus, VCs like the above can be efficiently validated by SMT solvers [24]. In this case, the solver will reject the above VC as *invalid* meaning the implication, and hence, the relevant subtyping requirement does not hold. So the refinement type system will *reject* `bad`.

On the other hand, a refinement system *accepts* `good`. Here, `+`'s type exactly captures its behaviour into the logic:

```
(+) :: x:Int → y:Int → {v:Int | v = x + y}
```

Thus, we can conclude that the divisor `z` is a positive number. The subtyping query for the

argument to `div` is

$$\texttt{x}{:}\{\texttt{x} \geq 0\},\ \texttt{y}{:}\{\texttt{y} \geq 0\},\ \texttt{z}{:}\{\texttt{z} = \texttt{y}+1\} \vdash \{\texttt{v} = \texttt{y}+1\} \preceq \{\texttt{v} > 0\}$$

which reduces to the *valid* VC

$$(\texttt{x} \geq 0) \wedge (\texttt{y} \geq 0) \wedge (\texttt{z} = \texttt{y}+1) \Rightarrow (\texttt{v} = \texttt{y}+1) \Rightarrow (\texttt{v} > 0)$$

2.1.2 Lazy Evaluation Makes VCs Unsound

To generate the classical VC, we ignored the "x is a value" guard that appears in the embedding of a binding ($\lfloor x{:}\tau \rfloor$) (Figure 2.1). Under lazy evaluation, ignoring this "is a value" guard can lead to unsoundness. Consider

```
diverge   :: Int → {v:Int | false}
diverge n = diverge n
```

The output type captures the *post-condition* that the function returns an `Int` satisfying `false`. This counter-intuitive specification states, in essence, that the function *does not terminate*, *i.e.* does not return *any* value. Any standard refinement type checker (or Floyd-Hoare verifier like Dafny [54]) will verify the given signature for `diverge` via the classical method of inductively *assuming* the signature holds for `diverge` and then *guaranteeing* the signature [39, 70]. Next, consider the call to `div` in `explode`:

```
explode   :: Int → Int
explode x = let {n = diverge 1; y = 0}
            in  x `div` y
```

To analyze `explode`, the refinement type system will check that `y` has type `Pos` at the call to `div`, *i.e.* will check that

$$\texttt{n}{:}\{\texttt{False}\},\ \texttt{y}{:}\{\texttt{y} = 0\} \vdash \{\texttt{v} = 0\} \preceq \{\texttt{v} > 0\} \tag{2.1}$$

In the environment, `n` is bound to the type corresponding to the *output* type of `diverge` and `y` is bound to the type stating `y` equals `0`. In this environment, we must prove that actual parameter's type – *i.e.* that of `y` – is a subtype of `Pos`. The subtyping, using the embedding of Figure 2.1 and ignoring the "is a value" guard, reduces to the VC:

$$\texttt{False} \wedge \texttt{y} = 0 \;\Rightarrow (\texttt{v} = 0) \Rightarrow (\texttt{v} > 0) \tag{2.2}$$

The SMT solver proves this VC valid using the contradiction in the antecedent, thereby unsoundly proving the call to `div` safe!

Eager vs. Lazy Verification Conditions We pause to emphasize that the problem lies in the fact that the classical technique for encoding subtyping (or generally, Hoare's "rule of consequence" [39]) with VCs is *unsound under lazy evaluation*. To see this, observe that the VC (2.2) is perfectly *sound* under eager (strict, call-by-value) evaluation. In the eager setting, the program is safe in that `div` is never called with the divisor `0`, as it is not called at all! The inconsistent antecedent in the VC logically encodes the fact that, under eager evaluation, the call to `div` is *dead code*. Of course, this conclusion is spurious under Haskell's lazy semantics. As `n` is not required, the program will dive headlong into evaluating the `div` and hence crash, rendering the VC meaningless.

The Problem is Laziness Readers familar with fully dependently typed languages, for instance Cayenne [2], Agda [71], Coq [8], and Idris [13], may be tempted to attribute the unsoundness to the presence of arbitrary recursion and hence non-termination (*e.g.* `diverge`). While it *is* possible to define a sound semantics for dependent types that mention potentially non-terminating expressions [51], it is not clear how to reconcile such semantics with decidable type checking.

Refinement type systems avoid this situation by carefully restricting types so that they do not contain arbitrary terms (even through substitution), but rather only terms from restricted logics that preclude arbitrary user-defined functions [106, 26, 88]. Very much like previous work, for now, we enforce the same restriction with a *well-formedness condition* on refinements (WF-BASE-D in Fig. 2.6). In Chapter 5 we present how our logic is extended with provably

terminating arbitrary terms, while preserving both soundness and decidability.

However, we show that restricting the logic of refinements *is plainly not sufficient for soundness* when laziness is combined with non-termination, as binders can be bound to diverging expressions. Unsurprisingly, in a strongly normalizing language the question of lazy or strict semantics is irrelevant for soundness, and hence an "easy" way to solve the problem would be to completely eliminate non-termination and rely on the soundness of previous refinement or dependent type systems! Instead, we show here how to recover soundness for a lazy language *without* imposing such a drastic requirement.

2.1.3 Semantics, Subtyping & Verification Conditions

To understand the problem, let us take a step back to get a clear view of the relationship between the operational semantics, subtyping, and verification conditions. We use the formulation of evaluation-order independent refinement subtyping developed for λ^H [51] in which refinements r are *arbitrary* expressions e from the source language. We define a denotation for types and use it to define subtyping declaratively.

Denotations of Types and Environments Recall the type `Pos` defined as `{v:Int | 0 < v}`. Intuitively, `Pos` denotes the *set of* `Int` expressions which evaluate to values greater than `0`. We formalize this intuition by defining the denotation of a type as:

$$[\![\{x{:}\tau \mid r\}]\!] \doteq \{e \mid \emptyset \vdash e : \tau, \text{ if } e \hookrightarrow^* v \text{ then } r[v/x] \hookrightarrow^* \texttt{True}\}$$

That is, the type denotes the set of expressions e that have the corresponding base type τ which *diverge or* reduce to values that make the refinement reduce to `True`. The guard $e \hookrightarrow^* v$ is crucially required to prove soundness in the presence of recursion. Thus, quoting [51], "refinement types specify partial and not total correctness".

An *environment* Γ is a sequence of type bindings and a *closing substitution* θ is a sequence of expression bindings:

$$\Gamma \doteq x_1{:}\tau_1, \ldots x_n{:}\tau_n \qquad\qquad \theta \doteq x_1 \mapsto e_1, \ldots, x_n \mapsto e_n$$

Thus, we define the denotation of Γ as the set of substitutions:

$$[\![\Gamma]\!] \doteq \{\theta \mid \forall x{:}\tau \in \Gamma.\theta(x) \in [\![\theta(\tau)]\!]\}$$

Declarative Subtyping Equipped with interpretations for types and environments, we define the *declarative subtyping* $\preceq$-BASE (over basic types b, shown in Figure 2.1) to be containment between the types' denotations:

$$\frac{\forall \theta \in [\![\Gamma]\!].\ [\![\theta(\{v{:}B \mid r_1\})]\!] \subseteq [\![\theta(\{v{:}B \mid r_2\})]\!]}{\Gamma \vdash \{v{:}B \mid r_1\} \preceq \{v{:}B \mid r_2\}}\ \preceq\text{-BASE}$$

Let us revisit the `explode` example from §2.1.2; recall that the function is safe under eager evaluation but unsafe under lazy evaluation. Let us see how the declarative subtyping allows us to reject in the one case and accept in the other.

Declarative Subtyping with Lazy Evaluation Let us revisit the query (2.1) to see whether it holds under the declarative subtyping rule $\preceq$-BASE. The denotation containment

$$\forall \theta \in [\![\mathtt{n}{:}\{\mathtt{False}\},\ \mathtt{y}{:}\{\mathtt{y} = 0\}]\!].[\![\theta\ \{\mathtt{v} = 0\}]\!] \subseteq [\![\theta\ \{\mathtt{v} > 0\}]\!] \tag{2.3}$$

does not hold. To see why, consider a θ that maps n to any diverging expression of type `Int` and y to the value 0. Then, $0 \in [\![\theta\ \{\mathtt{v} = 0\}]\!]$ but $0 \notin [\![\theta\ \{\mathtt{v} > 0\}]\!]$, thereby showing that the denotation containment does not hold.

Declarative Subtyping with Eager Evaluation Since denotational containment (2.3) does not hold, λ^H cannot verify `explode` under eager evaluation. However, Belo *et al.* [6] note that under eager (call-by-value) evaluation, each binder in the environment is only added *after* the previous binders have been reduced to *values*. Hence, under eager evaluation we can *restrict the range* of the closing substitutions to values (as opposed to expressions). Let us reconsider (2.3) in this new light: there *is no value* that we can map `n` to, so the set of denotations of the environment is empty. Hence, the containment (2.3) vacuously holds under eager evaluation, which proves the program safe. Belo's observation is implicitly used by refinement types for eager languages to

prove that the standard (*i.e.* call-by-value) reduction from subtyping to VC is sound.

Algorithmic Subtyping via Verification Conditions The above subtyping ($\preceq$-BASE) rule allows us to prove preservation and progress [51] but quantifies over evaluation of arbitrary expressions, and so is undecidable. To make checking *algorithmic* we approximate the denotational containment using *verification conditions* (VCs), formulas drawn from a decidable logic, that are valid only if the undecidable containment holds. As we have seen, the classical VC is sound only under eager evaluation. Next, let us use the distinctions between lazy and eager declarative subtyping, to obtain both sound and decidable VCs for the lazy setting.

Step 1: Restricting Refinements To Decidable Logics Given that λ^H refinements can be *arbitrary* expressions, the first step towards obtaining a VC, regardless of evaluation order, is to restrict the refinements to a *decidable* logic. We choose the quantifier free logic of equality, uninterpreted functions and linear arithmetic (QF-EUFLIA). Our typing rules ensure that for any valid derivation, all refinements belong in this restricted language.

Step 2: Translating Containment into VCs Our goal is to encode the denotation containment antecedent of $\preceq$-BASE

$$\forall \theta \in [\![\Gamma]\!].\ [\![\theta(\{v{:}B \mid r_1\})]\!] \subseteq [\![\theta(\{v{:}B \mid r_2\})]\!] \tag{2.4}$$

as a logical formula, that is valid *only when* the above holds. Intuitively, we can think of the closing substitutions θ as corresponding to *assignments* or *interpretations* ($|\theta|$) of variables X of the VC. We use the variable x to approximate denotational containment by stating that if x belongs to the type $\{v{:}B \mid r_1\}$ then x belongs to the type $\{v{:}B \mid r_2\}$:

$$\forall X \in dom(\Gamma), x.(|\Gamma|) \Rightarrow (|x{:}\{v{:}B \mid r_1\}|) \Rightarrow (|x{:}\{v{:}B \mid r_2\}|)$$

where $(|\Gamma|)$ and $(|x{:}\tau|)$ are respectively the translation of the environment and bindings into logical formulas that are only satisfied by assignments $(|\theta|)$ as shown in Figure 2.1. Using the translation

of bindings, and by renaming x to v, we rewrite the condition as

$$\forall X \in dom(\Gamma), v.(|\Gamma|) \Rightarrow (\text{“}v \text{ is a value”} \Rightarrow r_1) \Rightarrow (\text{“}v \text{ is a value”} \Rightarrow r_2)$$

Type refinements are carefully chosen to belong to the decidable logical sublanguage QF-EUFLIA, so we directly translate type refinements into the logic. Thus, what is left is to translate into logic the environment and the “is a value” guards. We postpone translation of the guards as we approximate the above formula by a *stronger*, *i.e.* sound with respect to 2.4, VC that just omits the guards:

$$\forall X \in dom(\Gamma), v.(|\Gamma|) \Rightarrow r_1 \Rightarrow r_2$$

To translate environments, we conjoin their bindings' translations:

$$(|x_1{:}\tau_1, \ldots, x_n{:}\tau_n|) \doteq (|x_1{:}\tau_1|) \wedge \ldots \wedge (|x_n{:}\tau_n|)$$

However, since types denote *partial correctness*, the translations must also explicitly account for possible divergence:

$$(|x{:}\{v{:}\texttt{Int} \mid r\}|) \doteq \text{“}x \text{ is a value”} \Rightarrow r[x/v]$$

That is, we *cannot* assume that each x satisfies its refinement r; we must *guard* that assumption with a predicate stating that x is bound to a value (not a diverging term).

The crucial question is: *how* can one discharge these guards to conclude that x indeed satisfies r? One natural route is to enrich the refinement logic with a predicate that states that “x is a value”, and then use the SMT solver to *explicitly* reason about this predicate and hence, divergence. Unfortunately, we show in §2.6, that such predicates lead to three-valued logics, which fall outside the scope of the efficiently decidable theories supported by current solvers. Hence, this route is problematic if we want to use existing SMT machinery to build automated verifiers for Haskell.

2.1.4 Our Answer: Implicit Reasoning About Divergence

One way forward is to *implicitly* reason about divergence by *eliminating* the "x is a value" guards (*i.e. value guards*) from the VCs.

Implicit Reasoning: Eager Evaluation Under eager evaluation the domain of the closing substitutions can be restricted to values [6]. Thus, we can trivially eliminate the value guards, as they are guaranteed to hold by virtue of the evaluation order. Returning to `explode`, we see that after eliminating the value guards, we get the VC (2.2) which is, therefore, sound under eager evaluation.

Implicit Reasoning: Lazy Evaluation However, with lazy evaluation, we cannot just eliminate the value guards, as the closing substitutions are not restricted to just values. Our solution is to take this reasoning out of the hands of the SMT logic and place it in the hands of a *stratified type system*. We use a non-deterministic β-reduction (formally defined in §2.2) to label each type as: A Div-type, written τ, which are the default types given to binders that *may diverge*, or, a Wnf-type, written $\tau^{\downarrow}$, which are given to binders that are guaranteed to reduce, in a finite number of steps, to *Haskell values* in Weak Head Normal Form (WHNF). Up to now we only discussed `Int` basic types, but our theory supports user-defined algebraic data types. An expression like `0 : repeat 0` is an infinite Haskell value. As we shall discuss, such infinite values cannot be represented in the logic. To distinguish infinite from finite values, we use a Fin-type, written $\tau^{\Downarrow}$, to label binders of expressions that are guaranteed to reduce to *finite values* with no redexes. This stratification lets us generate VCs that are sound for lazy evaluation. Let B be a basic labelled type. The key piece is the translation of environment bindings:

$$(\!|x{:}\{v{:}B \mid r\}|\!) \doteq \begin{cases} \texttt{True}, & \text{if } B \text{ is a Div type} \\ r[x/v], & \text{otherwise} \end{cases}$$

That is, if the binder may diverge, we simply *omit* any constraints for it in the VC, and otherwise the translation directly states (*i.e.* without the value guard) that the refinement holds. Returning to

`explode`, the subtyping query (2.1) yields the *invalid* VC

$$\texttt{True} \Rightarrow \texttt{v} = 0 \Rightarrow \texttt{v} > 0$$

and so `explode` is soundly rejected under lazy evaluation.

As binders appear in refinements and binders may refer to potentially infinite computations (*e.g.* `[0..]`), we must ensure that refinements are well defined (*i.e.* do not diverge). We achieve this via stratification itself, *i.e.* by ensuring that all refinements have type $\texttt{Bool}^{\Downarrow}$. By Corollary 1, this suffices to ensure that all the refinements are indeed well-defined and converge.

2.1.5 Verification With Stratified Types

While it is reassuring that the lazy VC soundly *rejects* unsafe programs like `explode`, we demonstrate by example that it usefully *accepts* safe programs. First, we show how the basic system – all terms have Div types – allows us to prove "partial correctness" properties without requiring termination. Second, we extend the basic system by using Haskell's pattern matching semantics to assign the pattern match scrutinees Wnf types, thereby increasing the expressiveness of the verifier. Third, we further improve the precision and usability of the system by using a termination checker to assign various terms Fin types. Fourth, we close the loop, by illustrating how the termination checker can itself be realized using refinement types. Finally, we use the termination checker to ensure that all refinements are well-defined (*i.e.* do converge).

Example: VCs and Partial Correctness The first example illustrates how, unlike Curry-Howard based systems, refinement types *do not require* termination. That is, we retain the Floyd-Hoare notion of "partial correctness" and can verify programs where *all* terms have Div-types. Consider `ex1` which uses the result of `collatz` as a divisor.

```
ex1   :: Int → Int
ex1 n = let x = collatz n in 10 `div` x

collatz :: Int → {v:Int | v = 1}
collatz n
  | n == 1    = 1
```

```
  | even n    = collatz (n / 2)
  | otherwise = collatz (3*n + 1)
```

The jury is still out on *whether* the `collatz` function terminates[1], but it is easy to verify that its output is a Div `Int` equal to `1`. At the call to `div` the parameter `x` has the output type of `collatz`, yielding the subtyping query:

$$\texttt{x}{:}\{\texttt{v}{:}\texttt{Int} \mid \texttt{v} = 1\} \vdash \{\texttt{v} = 1\} \preceq \{\texttt{v} > 0\}$$

where the sub-type is just the type of `x`. As `Int` is a Div type, the above reduces to the VC $(\texttt{True} \Rightarrow \texttt{v} = 1 \Rightarrow \texttt{v} > 0)$ which the SMT solver proves valid, verifying `ex1`.

Example: Improving Precision By Forcing Evaluation If all binders in the environment have Div-types then, effectively, the verifier can make *no* assumptions about the context in which a term evaluates, which leads to a drastic loss of precision. Consider:

```
ex2 = let {x = 1; y = inc x} in 10 `div` y

inc :: z:Int → {v:Int | v > z }
inc = \z → z + 1
```

The call to `div` in `ex2` is obviously safe, but the system would reject it, as the call yields the subtyping query:

$$\texttt{x}{:}\{\texttt{x}{:}\texttt{Int} \mid \texttt{x} = 1\},\ \texttt{y}{:}\{\texttt{y}{:}\texttt{Int} \mid \texttt{y} > \texttt{x}\} \vdash \{\texttt{v} > \texttt{x}\} \preceq \{\texttt{v} > 0\}$$

Which, as `x` is a Div type, reduces to the invalid VC:

$$\texttt{True} \Rightarrow \texttt{v} > \texttt{x} \Rightarrow \texttt{v} > 0$$

We could solve the problem by forcing evaluation of `x`. In Haskell the `seq` operator or a bang-pattern can be used to force evaluation. In our system the same effect is achieved by the `case-of` primitive: inside each case the matched binder is guaranteed to be a Haskell value in WHNF. This

[1] Collatz Conjecture: http://en.wikipedia.org/wiki/Collatz_conjecture

intuition is formalized by the typing rule (T-CASE-D), which checks each case after assuming the scrutinee and the match binder have Wnf types.

If we force x's evaluation, using the case primitive, the call to `div` yields the subtyping query:

$$\texttt{x}{:}\{\texttt{x}{:}\texttt{Int}^{\downarrow} \mid \texttt{x} = 1\},\ \texttt{y}{:}\{\texttt{y}{:}\texttt{Int} \mid \texttt{y} > \texttt{x}\} \vdash\ \{\texttt{v} > \texttt{x}\} \preceq \{\texttt{v} > 0\} \tag{2.5}$$

As x is Wnf, we accept `ex2` by proving the validity of the VC:

$$\texttt{x} = 1 \Rightarrow\ \texttt{v} > \texttt{x} \Rightarrow \texttt{v} > 0 \tag{2.6}$$

Example: Improving Precision By Termination While forcing evaluation allows us to ensure that certain environment binders have non-Div types, it requires program rewriting using case-splitting or the `seq` operator which leads to non-idiomatic code.

Instead, our next key optimization is based on the observation that in practice, *most terms don't diverge*. Thus, we can use a termination analysis to aggressively assign terminating expressions Fin types, which lets us strengthen the environment assumptions needed to prove the VCs. For example, in the `ex2` example the term `1` obviously terminates. Hence, we type x as $\texttt{Int}^{\Downarrow}$, yielding the subtyping query for `div` application:

$$\texttt{x}{:}\{\texttt{x}{:}\texttt{Int}^{\Downarrow} \mid \texttt{x} = 1\},\ \texttt{y}{:}\{\texttt{y}{:}\texttt{Int} \mid \texttt{y} > \texttt{x}\} \vdash\ \{\texttt{v} > \texttt{x}\} \preceq \{\texttt{v} > 0\} \tag{2.7}$$

As x is Fin, we accept `ex2` by proving the validity of the VC:

$$\texttt{x} = 1 \Rightarrow\ \texttt{v} > \texttt{x} \Rightarrow \texttt{v} > 0 \tag{2.8}$$

Example: Verifying Termination With Refinements While it is straightforward to conclude that the term `1` does not diverge, how do we do so in general? For example:

```
ex4 = let {x = f 9; y = inc x} in 10 `div` y
```

```
f   :: Nat → {v:Int | v = 1}
f n = if n == 0 then 1 else f (n-1)
```

We check the call to `div` via subtyping query (2.7) and VC (2.8), which requires us to prove that `f` terminates on *all* $\texttt{Nat}^{\Downarrow}$ inputs.

We solve this problem by showing how refinement types may themselves be used to prove termination, by following the classical recipe of proving termination via decreasing metrics [93] as embodied in sized types [41, 105]. The key idea is to show that each recursive call is made with arguments of a *strictly smaller* size, where the size is itself a well founded metric, *e.g.* a natural number.

We formalize this intuition by type checking recursive procedures in a termination-weakened environment where the procedure itself may only be called with arguments that are strictly smaller than the current parameter (using terminating fixpoints of §2.3.2). For example, to prove `f` terminates, we check its body in an environment

$$\texttt{n} : \texttt{Nat}^{\Downarrow}, \qquad \texttt{f} : \{\texttt{n}'{:}\texttt{Nat}^{\Downarrow} \mid \texttt{n}' < \texttt{n}\} \rightarrow \{\texttt{v} = 1\}$$

where we have weakened the type of `f` to stipulate that it *only* be (recursively) called with `Nat` values $\texttt{n}'$ that are *strictly less than* the (current) parameter `n`. The argument of `f` exactly captures these constraints, as using the Abbreviations of Figure 2.1 the argument of `f` is expanded to $\{\texttt{n}'{:}\texttt{Int}^{\Downarrow} \mid \texttt{n}' < \texttt{n} \wedge \texttt{n}' >= 0\}$. The body type-checks as the recursive call generates the valid VC:

$$0 \leq \texttt{n} \wedge \neg(0 = \texttt{n}) \Rightarrow \texttt{v} = \texttt{n} - 1 \Rightarrow (0 \leq \texttt{v} < \texttt{n})$$

Example: Diverging Refinements Finally, we discuss why refinements should always converge and how we statically ensure convergence. Consider the invalid specification

```
diverge 0 :: {v:Int | v = 12}
```

that states that the value of a diverging integer is `12`. The above specification should be rejected, as the refinement $\texttt{v} = 12$ does not evaluate to `True` ($\texttt{diverge}\ 0 = 12 \not\hookrightarrow^{*} \texttt{True}$), instead it diverges.

Definition	def	$::=$	$\texttt{measure}\ f :: \tau$ $eq_1 \ldots eq_n$
Equation	eq	$::=$	$f\ (D\ \overline{x}) = r$
Equation to Type	$(\lvert f\ (D\ \overline{x}) = r \rvert)$	$\doteq$	$D :: \overline{x{:}\tau} \rightarrow \{\texttt{v}{:}\tau \mid f\ \texttt{v} = r\}$

Figure 2.2. Syntax of Measures.

We want to check the validity of the formula $\texttt{v} = \texttt{12}$ under a model that maps `v` to the diverging integer `diverge 0`. Any system that decides this formula to be true will be unsound, *i.e.* the VCs will not soundly approximate subtyping. For similar reasons, the system should not decide that this formula is false. To reason about diverging refinements one needs three valued logic, where logical formulas can be solved to true, false, or diverging. Since we want to discharge VC using SMT solvers that currently do not support three valued reasoning, we exclude diverging refinements from types. To do so, we restrict $=$ to finite integers

$$(=) :: \texttt{Int}^{\Downarrow} \rightarrow \texttt{Int}^{\Downarrow} \rightarrow \texttt{Bool}^{\Downarrow}$$

and we say that $\{v{:}B \mid r\}$ is well-formed *iff* r has a $\texttt{Bool}^{\Downarrow}$ type (Corollary 1). Thus the initial invalid specification will be rejected as non well-formed.

2.1.6 Measures: From Integers to Data Types

So far, all our examples have used only integer and boolean expressions in refinements. To describe properties of algebraic data types, we use *measures*, introduced in prior work on Liquid Types [47]. Measures are inductively defined functions that can be used in refinements and provide an efficient way to axiomatize properties of data types. For example, `emp` determines whether a list is empty:

```
measure emp  :: [Int] → Bool
  emp []     = true
  emp (x:xs) = false
```

The syntax for measures deliberately looks like Haskell, but it is *far* more restricted, and should

really be considered as a separate language. A measure has exactly one argument and is defined by a list of equations, each of which has a simple pattern on the left hand side (Figure 2.2). The right-hand side of the equation is a refinement expression r. Measure definitions are typechecked in the usual way; we omit the typing rules which are standard. (Our metatheory does not support type polymorphism, so here we simply reason about lists of integers; however, our implementation supports polymorphism).

Denotational semantics The denotational semantics of types in λ^H in §2.1.3 is readily extended to support measures. In λ^H a refinement r is an arbitrary expression and calls to a measure are evaluated in the usual way by pattern matching. For example, with the above definition of `emp` it is straightforward to show that

$$[1,2,3] :: \{\mathtt{v}{:}[\mathtt{Int}] \mid \mathtt{not}\ (\mathtt{emp}\ \mathtt{v})\} \tag{2.9}$$

as the refinement `not (emp ([1, 2, 3]))` evaluates to `True`.

Measures as Axioms How can we reason about invocations of measures in the decidable logic of VCs? A natural approach is to treat a measure like `emp` as an uninterpreted function and add logical axioms that capture its behaviour. This looks easy: each equation of the measure definition corresponds to an axiom, thus:

$$\mathtt{emp}\ [] = \mathtt{True}$$

$$\forall \mathtt{x}, \mathtt{xs}.\, \mathtt{emp}\ (\mathtt{x} : \mathtt{xs}) = \mathtt{False}$$

Under these axioms the judgement 2.9 is indeed valid.

Measures as Refinements in Types of Data Constructors Axiomatizing measures is *precise*; that is, the axioms exactly capture the meaning of measures. Alas, axioms render SMT solvers *inefficient*, and render the VC mechanism *unpredictable*, as one must rely on various brittle syntactic matching and instantiation heuristics [25].

Instead, we use a different approach that is *both* precise *and* efficient. The key idea is

this: *instead of translating each measure equation into an axiom, we translate each equation into a refined type for the corresponding data constructor* [47]. This translation is given in Figure 2.2. For example, the definition of the measure `emp` yields the following refined types for the list data constructors:

$$\begin{array}{lcl} \texttt{[]} & :: & \{\texttt{v:[Int]} \mid \texttt{emp v} = \texttt{true}\} \\ \texttt{:} & :: & \texttt{x:Int} \rightarrow \texttt{xs:[Int]} \rightarrow \{\texttt{v:[Int]} \mid \texttt{emp v} = \texttt{false}\} \end{array}$$

These types ensure that: (1) each time a list value is *constructed*, its type carries the appropriate emptiness information. Thus our system is able to statically decide that (2.9) is valid and (2) each time a list value is *matched*, the appropriate emptiness information is used to improve precision of pattern matching, as we see next.

Using Measures As an example, we use the measure `emp` to provide an appropriate type for the `head` function:

```
head    :: {v:[Int] | not (emp v)} → Int
head xs = case xs of
            (x:_) → x
            []    → error "yikes"

error   :: {v:String | false} → a
error   = undefined
```

`head` is safe as its input type stipulates that it will only be called with lists that are *not* `[]`, and so `error "..."` is dead code. The call to `error` generates the subtyping query

$$\texttt{xs:}\{\texttt{xs:[Int]}^{\downarrow} \mid \neg(\texttt{emp xs})\},\ \texttt{b:}\{\texttt{b:[Int]}^{\downarrow} \mid (\texttt{emp xs}) = \texttt{true}\} \vdash \{\texttt{True}\} \preceq \{\texttt{False}\}$$

The match-binder `b` holds the result of the match [87]. In the `[]` case, we assign it the refinement of the type of `[]` which is $(\texttt{emp xs}) = \texttt{True}$. Since the call is done inside a `case-of` expressions both `xs` and `b` are in WHNF, thus they have Wnf types.

The verifier *accepts* the program as the above subtyping reduces to the valid VC:

$$\neg(\texttt{emp xs}) \wedge ((\texttt{emp xs}) = \texttt{True}) \Rightarrow \texttt{True} \Rightarrow \texttt{False}$$

Thus, our system supports idiomatic Haskell, *e.g.* taking the `head` of an infinite list:

```
ex x     = head (repeat x)

repeat   :: Int → {v:[Int] | not (emp v)}
repeat y = y : repeat y
```

Multiple Measures If a type has multiple measures, we simply refine each data constructor's type with the *conjunction* of the refinements from each measure. For example, consider a measure that computes the length of a list:

```
measure len  :: [Int] → Int
  len ([])   = 0
  len (x:xs) = 1 + len xs
```

Using the translation of Figure 2.2, we get the following types for list's data constructors.

$$[] :: \{\texttt{v:}[\texttt{Int}] \mid \texttt{len v} = 0\}$$

$$: :: \texttt{x:Int} \rightarrow \texttt{xs:}[\texttt{Int}] \rightarrow \{\texttt{v:}[\texttt{Int}] \mid \texttt{len v} = 1 + (\texttt{len xs})\}$$

The final types for list data are the conjunction of the refinements from `len` and `emp`:

$$[] :: \{\texttt{v:}[\texttt{Int}] \mid \texttt{emp v} = \texttt{true} \wedge \texttt{len v} = 0\}$$

$$: :: \texttt{x:Int} \rightarrow \texttt{xs:}[\texttt{Int}] \rightarrow \{\texttt{v:}[\texttt{Int}] \mid \texttt{emp v} = \texttt{false} \wedge \texttt{len v} = 1 + (\texttt{len xs})\}$$

$$
\begin{array}{rrcl}
\textbf{\textit{Constants}} & c & ::= & 0,1,-1,\ldots \mid \texttt{True},\texttt{False} \\
 & & \mid & +,-,\ldots \mid =,<,\ldots \mid \texttt{Crash} \\
\textbf{\textit{Values}} & v & ::= & c \mid \lambda x.e \mid D\,\overline{e} \\
\textbf{\textit{Expressions}} & e & ::= & v \mid x \mid e\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \\
 & & \mid & \mathsf{case}\ (x = e)\ \mathsf{of}\ \{D\,\overline{x}\ \rightarrow\ e\} \\
\textbf{\textit{Refinements}} & r & ::= & e \\
\textbf{\textit{Basic Types}} & B & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{T} \\
\textbf{\textit{Types}} & \tau & ::= & \{v{:}B \mid r\} \mid x{:}\tau \rightarrow \tau \\
\textbf{\textit{Contexts}} & C & ::= & \bullet \mid C\ e \mid c\ C \mid D\,\overline{e}\,C\,\overline{e} \\
 & & \mid & \mathsf{case}\ (x = C)\ \mathsf{of}\ \{D\,\overline{y}\ \rightarrow\ e\}
\end{array}
$$

Reduction $\boxed{e \hookrightarrow e}$

$$
\begin{array}{rcll}
C[e] & \hookrightarrow & C[e'] & \text{if } e \hookrightarrow e' \\
c\ v & \hookrightarrow & \delta(c,v) & \\
(\lambda x.e)\ e_x & \hookrightarrow & e\,[e_x/x] & \\
\mathsf{let}\ x = e_x\ \mathsf{in}\ e & \hookrightarrow & e\,[e_x/x] & \\
\mathsf{case}\ (x = D_j\ \overline{e})\ \mathsf{of}\ \{D_i\ \overline{y_i}\ \rightarrow\ e_i\} & \hookrightarrow & e_j\,[D_j\ \overline{e}/x]\,[\overline{e}/\overline{y_j}] &
\end{array}
$$

Figure 2.3. Syntax and Operational Semantics of λ^U.

2.2 Declarative Typing: λ^U

Next, we formalize our stratified refinement type system, in two steps. First, we present a core calculus λ^U, with a general β-reduction semantics. We describe the syntax, operational semantics, and sound but undecidable declarative typing rules for λ^U. Second, in §2.3, we describe QF-EUFLIA, a subset of λ^U that forms a decidable logic of refinements and use it to obtain λ^D with decidable SMT-based algorithmic typing.

2.2.1 Syntax

Figure 2.3 summarizes the syntax of λ^U, which is essentially the calculus λ^H [51] *without* the dynamic checking features (like casts), but *with* the addition of data constructors. In λ^U, as in λ^H, refinement expressions r are not drawn from a decidable logical sublanguage, but can be arbitrary expressions e (hence $r ::= e$ in Figure 2.3). This choice allows us to prove preservation

and progress, but renders typechecking undecidable.

Constants The primitive constants of λ^U include `True`, `False`, `0`, `1`, -1, *etc.*, and arithmetic and logical operators like $+$, $-$, $\leq$,/, $\wedge$, $\neg$. In addition, we include a special *untypable* constant `Crash` that models "going wrong". Primitive operations return a `Crash` when invoked with inputs outside their domain, *e.g.* when / is invoked with `0` as the divisor or when `assert` is applied to `false`.

Data Constructors We encode data constructors as special constants. Each data type has an arity $\mathsf{Arity}(T)$ that represents the exact number of data constructors that return a value of type T. For example the data type $[\texttt{Int}]$, which represents lists of integers, has two data constructors: [] and :, *i.e.* has arity 2.

Values & Expressions The values of λ^U include constants, λ-abstractions $\lambda x.e$, and fully applied data constructors D that wrap expressions. The expressions of λ^U include values, as well as variables x, applications $e\ e$, and the `case` and `let` expressions.

2.2.2 Operational Semantics

Figure 2.3 summarizes the small step, contextual β-reduction semantics for λ^U. We allow for reductions under data constructors, and thus, values may be further reduced. We write $e \hookrightarrow^j e'$ if there exist $e_1, \ldots, e_j$ such that e is e_1, e' is e_j and $\forall i, j, 1 \leq i < j$, we have $e_i \hookrightarrow e_{i+1}$. We write $e \hookrightarrow^* e'$ if there exists a (finite) j such that $e \hookrightarrow^j e'$.

Constants Application of a constant requires the argument to be reduced to a value; in a single step the expression is reduced to the output of the primitive constant operation. For example, consider $=$, the primitive equality operator on integers. We have $\delta(=, n) \doteq =_n$ where $\delta(=_n, m)$ equals `True` iff m is the same as n.

2.2.3 Types

λ^U types include basic types, which are *refined* with predicates, and dependent function types. *Basic types B* comprise integers, booleans, and a family of data-types T (representing lists, trees *etc.*). For example, the data type $[\texttt{Int}]$ represents lists of integers. We refine basic types

with predicates (boolean valued expressions e) to obtain *basic refinement types* $\{v{:}B \mid e\}$. Finally, we have dependent *function types* $x{:}\tau_x \to \tau$ where the input x has the type τ_x and the output τ may refer to the input binder x.

Notation We write B to abbreviate $\{v{:}B \mid \texttt{True}\}$ and $\tau_x \to \tau$ to abbreviate $x{:}\tau_x \to \tau$ if x does not appear in τ. We use $\_$ for unused binders. We write $\{v{:}\texttt{Nat}^l \mid r\}$ to abbreviate $\{v{:}\texttt{Int}^l \mid 0 \leq v \wedge r\}$.

Denotations Each type τ *denotes* a set of expressions $[\![\tau]\!]$, that are defined via the dynamic semantics [51]. Let $\lfloor\tau\rfloor$ be the type we get if we erase all refinements from τ and $e{:}\lfloor\tau\rfloor$ be the standard typing relation for the typed lambda calculus. Then, we define the denotation of types as:

$$[\![\{x{:}\texttt{B} \mid r\}]\!] \doteq \{e \mid e{:}\texttt{B},\ \text{if } e \hookrightarrow^* w \text{ then } r[w/x] \hookrightarrow^* \texttt{True}\}$$

$$[\![x{:}\tau_x \to \tau]\!] \doteq \{e \mid e{:}\lfloor\tau_x \to \tau\rfloor, \forall e_x \in [\![\tau_x]\!].\ e\ e_x \in [\![\tau[e_x/x]]\!]\}$$

Constants For each constant c we define its type $\mathsf{Ty}(c)$ such that $c \in [\![\mathsf{Ty}(c)]\!]$, *e.g.*

$$\begin{array}{lcl}
\mathsf{Ty}(3) & \doteq & \{\texttt{v:Int} \mid \texttt{v} = 3\} \\
\mathsf{Ty}(+) & \doteq & \texttt{x:Int} \to \texttt{y:Int} \to \{\texttt{v:Int} \mid \texttt{v} = \texttt{x} + \texttt{y}\} \\
\mathsf{Ty}(/) & \doteq & \texttt{Int} \to \{\texttt{v:Int} \mid \texttt{v} > 0\} \to \texttt{Int} \\
\mathsf{Ty}(\texttt{error}_\tau) & \doteq & \{\texttt{v:Int} \mid \texttt{False}\} \to \tau
\end{array}$$

So, by definition we get the constant typing lemma.

Lemma 1. *[Constant Typing] For every constant c, $c \in [\![\mathsf{Ty}(c)]\!]$.*

Thus, if $\mathsf{Ty}(c) \doteq x{:}\tau_x \to \tau$, then for every value $w \in [\![\tau_x]\!]$, we require that $\delta(c,w) \in [\![\tau[w/x]]\!]$. For every value $w \notin [\![\tau_x]\!]$, it suffices to define $\delta(c,w)$ as `Crash`, a special untyped value.

Data Constructors The types of data constructor constants are refined with predicates that track the semantics of the *measures* associated with the data type. For example, as discussed in §2.1.6

Well-Formedness $\boxed{\Gamma \vdash_U \tau}$

$$\frac{\Gamma, v{:}B \vdash_U r : \texttt{Bool}}{\Gamma \vdash_U \{v{:}B \mid r\}}\ \textsc{WF-Base} \qquad \frac{\Gamma \vdash_U \tau_x \quad \Gamma, x{:}\tau_x \vdash_U \tau}{\Gamma \vdash_U x{:}\tau_x \to \tau}\ \textsc{WF-Fun}$$

Subtyping $\boxed{\Gamma \vdash_U \tau_1 \preceq \tau_2}$

$$\frac{\forall \theta \in [\![\Gamma]\!].[\![\theta(\{v{:}B \mid r_1\})]\!] \subseteq [\![\theta(\{v{:}B \mid r_2\})]\!]}{\Gamma \vdash_U \{v{:}B \mid r_1\} \preceq \{v{:}B \mid r_2\}}\ \preceq\textsc{-Base}$$

$$\frac{\Gamma \vdash_U \tau'_x \preceq \tau_x \quad \Gamma, x{:}\tau'_x \vdash_U \tau \preceq \tau'}{\Gamma \vdash_U x{:}\tau_x \to \tau \preceq x{:}\tau'_x \to \tau'}\ \preceq\textsc{-Fun}$$

Typing $\boxed{\Gamma \vdash_U e : \tau}$

$$\frac{(x,\tau) \in \Gamma}{\Gamma \vdash_U x : \tau}\ \textsc{T-Var} \qquad \frac{}{\Gamma \vdash_U c : \mathsf{Ty}(c)}\ \textsc{T-Con}$$

$$\frac{\Gamma \vdash_U e : \tau' \quad \Gamma \vdash_U \tau' \preceq \tau \quad \Gamma \vdash_U \tau}{\Gamma \vdash_U e : \tau}\ \textsc{T-Sub}$$

$$\frac{\Gamma, x{:}\tau_x \vdash_U e : \tau \quad \Gamma \vdash_U \tau_x}{\Gamma \vdash_U \lambda x.e : (x{:}\tau_x \to \tau)}\ \textsc{T-Fun} \qquad \frac{\Gamma \vdash_U e_1 : (x{:}\tau_x \to \tau) \quad \Gamma \vdash_U e_2 : \tau_x}{\Gamma \vdash_U e_1\ e_2 : \tau\,[e_2/x]}\ \textsc{T-App}$$

$$\frac{\Gamma \vdash_U e_x : \tau_x \quad \Gamma, x{:}\tau_x \vdash_U e : \tau \quad \Gamma \vdash_U \tau}{\Gamma \vdash_U \mathsf{let}\ x = e_x\ \mathsf{in}\ e : \tau}\ \textsc{T-Let}$$

$$\frac{\begin{array}{cc}\Gamma \vdash_U e : \{v{:}T \mid r\} & \Gamma \vdash_U \tau \\ \forall i.\mathsf{Ty}(D_i) = \overline{y_j{:}\tau_j} \to \{v{:}T \mid r_i\} & \Gamma, \overline{y_j{:}\tau_j}, x{:}\{v{:}T \mid r \wedge r_i\} \vdash_U e_i : \tau\end{array}}{\Gamma \vdash_U \mathsf{case}\ (x = e)\ \mathsf{of}\ \{D_i\ \overline{y_j}\ \to\ e_i\} : \tau}\ \textsc{T-Case}$$

Figure 2.4. Type checking of λ^U.

we use `emp` to refine the list data constructors' types:

$$\begin{array}{lcl}\mathsf{Ty}([]) & \doteq & \{\texttt{v}{:}[\texttt{Int}] \mid \texttt{emp v}\} \\ \mathsf{Ty}(:) & \doteq & \texttt{Int} \to [\texttt{Int}] \to \{\texttt{v}{:}[\texttt{Int}] \mid \neg(\texttt{emp v})\}\end{array}$$

By construction it is easy to prove that Lemma 1 holds for data constructors. For example, `emp []` goes to `True`.

2.2.4 Type Checking

Next, we present the type-checking judgments and rules of λ^U.

Environments and Closing Substitutions A *type environment* Γ is a sequence of type bindings $x_1{:}\tau_1, \ldots, x_n{:}\tau_n$. An environment denotes a set of *closing substitutions* θ which are sequences of expression bindings: $x_1 \mapsto e_1, \ldots, x_n \mapsto e_n$ such that:

$$[\![\Gamma]\!] \doteq \{\theta \mid \forall x{:}\tau \in \Gamma.\theta(x) \in [\![\theta(\tau)]\!]\}$$

Judgments We use environments to define three kinds of rules: Well-formedness, Subtyping, and Typing [51, 7]. A judgment $\Gamma \vdash_U \tau$ states that the refinement type τ is well-formed in the environment Γ. Intuitively, the type τ is well-formed if all the refinements in τ are `Bool`-typed in Γ. A judgment $\Gamma \vdash_U \tau_1 \preceq \tau_2$ states that the type τ_1 is a subtype of τ_2 in the environment Γ. Informally, τ_1 is a subtype of τ_2 if, when the free variables of τ_1 and τ_2 are bound to expressions described by Γ, the denotation of τ_1 is *contained in* the denotation of τ_2. Subtyping of basic types reduces to denotational containment checking. That is, for any closing substitution θ in the denotation of Γ, for every expression e, if $e \in [\![\theta(\tau_1)]\!]$ then $e \in [\![\theta(\tau_2)]\!]$. A judgment $\Gamma \vdash_U e : \tau$ states that the expression e has the type τ in the environment Γ. That is, when the free variables in e are bound to expressions described by Γ, the expression e will evaluate to a value described by τ.

Soundness In [101], we use the (undecidable) $\preceq$-BASE to prove that each step of evaluation preserves typing and that if an expression is not a value, then it can be further evaluated:

- **Preservation:** If $\emptyset \vdash_U e : \tau$ and $e \hookrightarrow e'$, then $\emptyset \vdash_U e' : \tau$.
- **Progress:** If $\emptyset \vdash_U e : \tau$ and $e \neq w$, then $e \hookrightarrow e'$.

We combine the above to prove that evaluation preserves typing and that a well typed term will not `Crash`.

Theorem 1. *[Soundness of λ^U]*

Expressions, ***Values***, ***Constants***, ***Basic types***: see Figure 2.3

Types	τ	::=	$\{v{:}\mathtt{B} \mid r\} \mid \{v{:}\mathtt{B}^l \mid r\}$
		$\mid$	$x{:}\tau \to \tau$
Labels	l	::=	$\downarrow \mid \Downarrow$
Refinements	r	::=	p
Predicates	p	::=	$p = p \mid p < p \mid p \wedge p \mid \neg p$
		$\mid$	$n \mid x \mid f\ \overline{p} \mid p \oplus p$
		$\mid$	$\mathtt{True} \mid \mathtt{False}$
Measures	f, g, h		
Operators	$\oplus$	::=	$+ \mid - \mid \ldots$
Integers	n	::=	$0 \mid 1 \mid -1 \mid \ldots$
Domain	d	::=	$n \mid c_w \mid D\ \overline{d} \mid \mathtt{True} \mid \mathtt{False}$
Model	σ	::=	$x_1 \mapsto d_1, \ldots, x_n \mapsto d_n$
Lifted Values	$w^\perp$	::=	$c \mid \lambda x.e \mid D\ \overline{w^\perp} \mid \perp$

Figure 2.5. Syntax of λ^D.

- ***Type-Preservation:*** *If* $\emptyset \vdash_U e : \tau$, $e \hookrightarrow^* w$ *then* $\emptyset \vdash_U w : \tau$.
- ***Crash-Freedom:*** *If* $\emptyset \vdash_U e : \tau$ *then* $e \not\hookrightarrow^* \mathtt{Crash}$.

We prove the above following the overall recipe of [51]. Crash-freedom follows from type-preservation, as `Crash` has no type. The Substitution Lemma, in particular, follows from a connection between the typing relation and type denotations:

Lemma 2. *[Denotation Typing] If* $\emptyset \vdash_U e : \tau$ *then* $e \in [\![\tau]\!]$.

2.3 Algorithmic Typing: λ^D

While λ^U is sound, it cannot be *implemented* thanks to the undecidable denotational containment rule $\preceq$-BASE (Figure 2.4). Next, we go from λ^U to λ^D, a core calculus with sound, SMT-based algorithmic type-checking in four steps. First, we show how to restrict the language of refinements to an SMT-decidable sub-language QF-EUFLIA (§2.3.1). Second, we *stratify* the types to specify whether their inhabitants may diverge, must reduce to values, or must reduce to

All rules as in Figure 5.3 except as follows:

Well-Formedness $\boxed{\Gamma \vdash_D \tau}$

$$\frac{\Gamma, v{:}B \vdash_D p : \texttt{Bool}^{\Downarrow}}{\Gamma \vdash_D \{v{:}B \mid p\}} \text{ WF-BASE-D}$$

Subtyping $\boxed{\Gamma \vdash_D \tau_1 \preceq \tau_2}$

$$\frac{(|\Gamma, v : B|) \Rightarrow (|p_1|) \Rightarrow (|p_2|) \text{ is valid}}{\Gamma \vdash_D \{v{:}B \mid p_1\} \preceq \{v{:}B \mid p_2\}} \preceq\text{-BASE-D}$$

Typing $\boxed{\Gamma \vdash_D e : \tau}$

$$\frac{\Gamma \vdash_D e_1 : (x{:}\tau_x \to \tau) \qquad \Gamma \vdash_D y : \tau_x}{\Gamma \vdash_D e_1\ y : \tau[y/x]} \text{ T-APP-D}$$

$$\frac{\begin{array}{ccc} l \notin \{\Downarrow, \downarrow\} \Rightarrow \tau \text{ is Div} & \Gamma \vdash_D e : \{v{:}T^l \mid r\} & \Gamma \vdash_D \tau \\ \forall i. \mathsf{Ty}(D_i) = \overline{y_j}\tau_j \to \{v{:}T \mid r_i\} & \Gamma, \overline{y_j{:}\tau_j}, x{:}\{v{:}T^{\downarrow} \mid r \wedge r_i\} \vdash_D e_i : \tau \end{array}}{\Gamma \vdash_D \mathsf{case}\ (x = e)\ \mathsf{of}\ \{D_i\ \overline{y_j}\ \to\ e_i\} : \tau} \text{ T-CASE-D}$$

Figure 2.6. Type checking of λ^D.

finite values (§2.3.2). Third, we show how to *enforce* the stratification by encoding recursion using special fixpoint combinator constants (§2.3.2). Finally, we show how to use QF-EUFLIA and the stratification to approximate the undecidable $\preceq$-BASE with a decidable verification condition $\preceq$-BASE-D, thereby obtaining the algorithmic system λ^D (§2.3.3).

2.3.1 Refinement Logic: QF-EUFLIA

Figure 2.5 summarizes the syntax of λ^D. Refinements r are now predicates p, drawn from QF-EUFLIA, the decidable logic of equality, uninterpreted functions and linear arithmetic [69]. Predicates p include linear arithmetic constraints, function application where function symbols correspond to measures (as described in §2.1.6), and boolean combinations of sub-predicates.

Well-Formedness For a predicate to be well-formed it should be boolean and arithmetic operators should be applied to integer terms, measures should be applied to appropriate arguments (*i.e.* `emp` is applied to [`Int`]), and equality or inequality to basic (integer or boolean) terms. Furthermore, we require that refinements, and thus measures, always evaluate to a value. We capture

these requirements by assigning appropriate types to operators and measure functions, after which we require that each refinement r has type $\texttt{Bool}^{\Downarrow}$ (rule WF-BASE-D in Figure 2.6).

Assignments Figure 2.5 defines the elements d of the domain $\mathscr{D}$ of integers, booleans, and data constructors that wrap elements from $\mathscr{D}$. The domain $\mathscr{D}$ also contains a constant c_w for each value w of λ^U that does not otherwise belong in $\mathscr{D}$ (*e.g.* functions or other primitives). An *assignment* σ is a map from variables to $\mathscr{D}$.

Satisfiability & Validity We interpret boolean predicates in the logic over the domain $\mathscr{D}$. We write $\sigma \models p$ if σ is a model of p. We omit the formal definition for space. A predicate p is *satisfiable* if there *exists* $\sigma \models p$. A predicate p is *valid* if *for all* assignments $\sigma \models p$.

Connecting Evaluation and Logic To prove soundness, we need to formally connect the notion of logical models with the evaluation of a refinement to `True`. We do this in several steps, briefly outlined for brevity (the detailed proof is in [101]). First, we introduce a primitive *bottom expression* $\bot$ that can have *any* Div type, but does not evaluate. Second, we define *lifted values* $w^{\bot}$ (Figure 2.5), which are values that contain $\bot$. Third, we define *lifted substitutions* $\theta^{\bot}$, which are mappings from variables to lifted values. Finally, we show how to *embed* a lifted substitution $\theta^{\bot}$ into a *set of* assignments $(|\theta^{\bot}|)$ where, intuitively speaking, each $\bot$ is replaced by some arbitrarily chosen element of $\mathscr{D}$. Now, we can connect evaluation and logical satisfaction:

Theorem 2. *If* $\emptyset \vdash_D \theta^{\bot}(p) : \texttt{Bool}^{\Downarrow}$, *then* $\theta^{\bot}(p) \hookrightarrow^* \texttt{True}$ *iff* $\forall \sigma \in (|\theta^{\bot}|).\sigma \models p$.

Restricting Refinements to Predicates Our goal is to restrict $\preceq$-BASE so that only predicates from the decidable logic QF-EUFLIA (not arbitrary expressions) appear in implications $(|\Gamma|) \Rightarrow \{v{:}b \mid p_1\} \Rightarrow \{v{:}b \mid p_2\}$. Towards this goal, as shown in Figures 2.5 and 2.6, we restrict the syntax and well-formedness of types to contain only predicates and we convert the program to ANF after which we can restrict the application rule T-APP-D to applications to variables, which ensures that refinements remain within the logic after substitution [79]. Recall, that this is not enough to ensure that refinements do converge, as under lazy evaluation, even binders can refer to potentially divergent values.

2.3.2 Stratified Types

The typing rules for λ^D are given in Figure 2.6. Instead of *explicitly* reasoning about divergence or strictness in the refinement logic, which leads to significant theoretical and practical problems, as discussed in §2.6, we choose to reason *implicitly* about divergence within the type system. Thus, the second critical step in our path to λ^D is the stratification of types into those inhabited by potentially diverging terms, terms that only reduce to values, and terms which reduce to finite values. Furthermore, the stratification crucially allows us to prove Theorem 2, which requires that refinements do not diverge (*e.g.* by computing the length of an infinite list) by ensuring that inductively defined measures are only applied to finite values. Next, we describe how we stratify types with labels and then type the various constants, in particular the fixpoint combinators, to enforce stratification.

Labels We specify stratification using two *labels* for types. The label $\downarrow$ (resp. $\Downarrow$) is assigned to types given to expressions that reduce (using β-reduction from Figure 2.3) to a value w (resp. *finite* value, *i.e.* an element of the inductively defined $\mathscr{D}$). Formally,

$$\textbf{\textit{Wnf types}} \quad [\![\{v{:}B^{\downarrow} \mid r\}]\!] \doteq [\![\{v{:}B \mid r\}]\!] \cap \{e \mid e \hookrightarrow^* w\} \tag{2.10}$$

$$\textbf{\textit{Fin types}} \quad [\![\{v{:}B^{\Downarrow} \mid r\}]\!] \doteq [\![\{v{:}B \mid r\}]\!] \cap \{e \mid e \hookrightarrow^* d\} \tag{2.11}$$

Unlabelled types are assigned to expressions that may diverge. Note that for any B and refinement r we have

$$[\![\{v{:}B^{\Downarrow} \mid r\}]\!] \subseteq [\![\{v{:}B^{\downarrow} \mid r\}]\!] \subseteq [\![\{v{:}B \mid r\}]\!]$$

The first two sets are *equal* for `Int` and `Bool`, and *unequal* for (lazily) constructed data types T. We need not stratify function types (*i.e.* they are Div types) as binders with function types do not appear inside the VC, and are not applied to measures.

Enforcing Stratification We enforce stratification in two steps. First, the T-CASE-D rule uses the operational semantics of case-of to type-check each case in an environment where the scrutinee x is assumed to have a Wnf type. All the other rules, not mentioned in Figure 2.6, remain the same

as in Figure 2.4. Second, we create stratified variants for the primitive constants and *separate* fixpoint combinator constants for (arbitary, potentially non-terminating) recursion (`fix`) and bounded recursion (`tfix`).

Stratified Primitives First, we restrict the primitive operators whose output types are refined with logical operators, so they are only invoked on finite arguments (so that the corresponding refinements are guaranteed to not diverge).

$$\begin{aligned}
\mathsf{Ty}(n) &\doteq \{v{:}\mathtt{Int}^{\Downarrow} \mid v = n\} \\
\mathsf{Ty}(=) &\doteq x{:}\text{T-VAR-BASE}^{\Downarrow} \to y{:}\text{T-VAR-BASE}^{\Downarrow} \to \{v{:}\mathtt{Bool}^{\Downarrow} \mid v \Leftrightarrow x = y\} \\
\mathsf{Ty}(+) &\doteq x{:}\mathtt{Int}^{\Downarrow} \to y{:}\mathtt{Int}^{\Downarrow} \to \{v{:}\mathtt{Int}^{\Downarrow} \mid v = x + y\} \\
\mathsf{Ty}(\wedge) &\doteq x{:}\mathtt{Bool}^{\Downarrow} \to y{:}\mathtt{Bool}^{\Downarrow} \to \{v{:}\mathtt{Bool}^{\Downarrow} \mid v \Leftrightarrow x \wedge y\}
\end{aligned}$$

It is easy to prove that the above primitives respect their stratification labels, *i.e.* belong in the denotations of their types.

Note that the above types are restricted in that they can only be applied to finite arguments. In future work 8, we could address this issue with unrefined versions of primitive types that soundly allow operation on arbitrary arguments. For example, with the current type for $+$, addition of potentially diverging expressions is rejected. Thus, we could define an unrefined signature

$$\mathsf{Ty}(+) \doteq x{:}\mathtt{Int} \to y{:}\mathtt{Int} \to \mathtt{Int}$$

and allow the two types of $+$ to co-exist (as an intersection type), where the type checker would choose the precise refined type if and only if both of $+$'s arguments are finite.

Diverging Fixpoints (*$\mathtt{fix}_\tau$*) Next, note that the only place where divergence enters the picture is through the fixpoint combinators used to encode recursion. For any function or basic type $\tau \doteq \tau_1 \to \ldots \to \tau_n$, we define the *result* to be the type τ_n.

For each τ whose result is a Div type, there is a *diverging fixpoint* combinator $\mathtt{fix}_\tau$, such

that

$$\delta(\texttt{fix}_\tau, f) \doteq f\ (\texttt{fix}_\tau\ f)$$
$$\mathsf{Ty}(\texttt{fix}_\tau) \doteq (\tau \to \tau) \to \tau$$

i.e., $\texttt{fix}_\tau$ yields recursive functions of type τ. Of course, $\texttt{fix}_\tau$ belongs in the denotation of its type [78] *only if* the result type is a Div type (and *not* when the result is a Wnf or Fin type). Thus, we restrict diverging fixpoints to functions with Div result types.

Indexed Fixpoints ($\texttt{tfix}^n_\tau$) For each type τ whose result is a Fin type, we have a family of *indexed* fixpoints combinators $\texttt{tfix}^n_\tau$:

$$\delta(\texttt{tfix}^n_\tau, f) \doteq \lambda m.f\ m\ (\texttt{tfix}^m_\tau\ f)$$
$$\mathsf{Ty}(\texttt{tfix}^n_\tau) \doteq (n{:}\texttt{Nat}^{\Downarrow} \to \tau_n \to \tau) \to \tau_n$$
$$\text{where, } \tau_n \doteq \{v{:}\texttt{Nat}^{\Downarrow} \mid v < n\} \to \tau$$

τ_n is a *weakened* version of τ that can only be invoked on inputs *smaller* than n. Thus, we enforce termination by requiring that $\texttt{tfix}^n_\tau$ is *only* called with m that are *strictly smaller than* n. As the indices are well-founded `Nat`s, evaluation will terminate.

Terminating Fixpoints ($\texttt{tfix}_\tau$) Finally, we use the indexed combinators to define the *terminating* fixpoint combinator $\texttt{tfix}_\tau$ as:

$$\delta(\texttt{tfix}_\tau, f) \doteq \lambda n.f\ n\ (\texttt{tfix}^n_\tau\ f)$$
$$\mathsf{Ty}(\texttt{tfix}_\tau) \doteq (n{:}\texttt{Nat}^{\Downarrow} \to \tau_n \to \tau) \to \texttt{Nat}^{\Downarrow} \to \tau$$

Thus, the top-level call to the recursive function requires a $\texttt{Nat}^{\Downarrow}$ parameter n that acts as a *starting* index, after which, all "recursive" calls are to combinators with *smaller* indices, ensuring termination.

Example: Factorial Consider the factorial function:

$$\texttt{fac} \doteq \lambda n.\lambda f.\texttt{case}\ \_ = (n = 0)\ \texttt{of}\ \left\{ \begin{array}{l} \texttt{True} \rightarrow 1 \\ \_ \rightarrow n \times f(n-1) \end{array} \right\}$$

Let $\tau \doteq \texttt{Nat}^{\Downarrow}$. We prove termination by typing

$$\emptyset \vdash_D \texttt{tfix}_\tau\ \texttt{fac} : \texttt{Nat}^{\Downarrow} \rightarrow \tau$$

To understand *why*, note that $\texttt{tfix}^n_\tau$ is only called with arguments strictly smaller than n

$$\begin{aligned} \texttt{tfix}_\tau\ \texttt{fac}\ n &\hookrightarrow^* \texttt{fac}\ n\ (\texttt{tfix}^n_\tau\ \texttt{fac}) \\ &\hookrightarrow^* n \times (\texttt{tfix}^n_\tau\ \texttt{fac}\ (n-1)) \\ &\hookrightarrow^* n \times (\texttt{fac}\ (n-1)\ (\texttt{tfix}^{n-1}_\tau\ \texttt{fac})) \\ &\hookrightarrow^* n \times n-1 \times (\texttt{tfix}^{n-1}_\tau\ \texttt{fac}\ (n-2)) \\ &\hookrightarrow^* n \times n-1 \times \ldots \times (\texttt{tfix}^1_\tau\ \texttt{fac}\ 0) \\ &\hookrightarrow^* n \times n-1 \times \ldots \times (\texttt{fac}\ 0\ (\texttt{tfix}^0_\tau\ \texttt{fac})) \\ &\hookrightarrow^* n \times n-1 \times \ldots \times 1 \end{aligned}$$

Soundness of Stratification To formally *prove* that stratification is soundly enforced, it suffices to prove that the Denotation Lemma 2 holds for λ^D. This, in turn, boils down to proving that each (stratified) constant belongs in its type's denotation, *i.e.* each $c \in [\![\mathsf{Ty}(c)]\!]$ or that the Lemma 1 holds for λ^D. The crucial part of the above is proving that the indexed and terminating fixpoints inhabit their types' denotations.

Theorem 3. *[Fixpoint Typing]*

- $\texttt{fix}_\tau \in [\![\mathsf{Ty}(\texttt{fix}_\tau)]\!]$,
- $\forall n.\texttt{tfix}^n_\tau \in [\![\mathsf{Ty}(\texttt{tfix}^n_\tau)]\!]$,

- $\texttt{tfix}_\tau \in [\![\mathsf{Ty}(\texttt{tfix}_\tau)]\!]$.

With the above we can prove soundness of Stratification as a corollary Denotation Lemma 2, given the interpretations of the stratified types.

Corollary 1. *[Soundness of Stratification]*

1. *If* $\emptyset \vdash_D e : \tau^{\Downarrow}$*, then evaluation of e is finite.*
2. *If* $\emptyset \vdash_D e : \tau^{\downarrow}$*, then e reduces to WHNF.*
3. *If* $\emptyset \vdash_D e : \{v{:}\tau \mid p\}$*, then p cannot diverge.*

Finally, as a direct implication the well-formedness rule WF-BASE-D we conclude 3, *i.e.* that refinements cannot diverge.

2.3.3 Verification With Stratified Types

We can put the pieces together to obtain an algorithmic implication rule $\preceq$-BASE-D instead of the undecidable $\preceq$-BASE (from Figure 2.4). Intuitively, each closing substitution θ will correspond to a set of logical assignments $(|\theta|)$. Thus, we will translate Γ into logical formula $(|\Gamma|)$ and denotation inclusion into logical implication such that:

- $\theta \in [\![\Gamma]\!]$ *iff* all $\sigma \in (|\theta|)$ satisfy $(|\Gamma|)$, and
- $\theta\{v{:}B \mid p_1\} \subseteq \theta\{v{:}B \mid p_2\}$ *iff* all $\sigma \in (|\theta|)$ satisfy $p_1 \Rightarrow p_2$.

Translating Refinements & Environments To translate environments into logical formulas, recall that $\theta \in [\![\Gamma]\!]$ iff for each $x{:}\tau \in \Gamma$, we have $\theta(x) \in [\![\theta(\tau)]\!]$. Thus,

$$(|x_1{:}\tau_1, \ldots, x_n{:}\tau_n|) \doteq (|x_1{:}\tau_1|) \wedge \ldots \wedge (|x_n{:}\tau_n|)$$

How should we translate a single binding? Since a binding denotes

$$[\![\{x{:}\texttt{B} \mid p\}]\!] \doteq \{e \mid \text{if } e \hookrightarrow^* w \text{ then } p\,[w/x] \hookrightarrow^* \texttt{True}\}$$

a direct translation would require a logical value predicate $\mathsf{Val}(x)$, which we could use to obtain the logical translation

$$(|\{x{:}\mathtt{B} \mid p\}|) \doteq \neg\mathsf{Val}(x) \vee p$$

This translation poses several theoretical and practical problems that preclude the use of existing SMT solvers (as detailed in §2.6). However, our stratification guarantees (cf. (2.10), (2.11)) that labeled types reduce to values and so we can simply conservatively translate the Div and labeled (Wnf, Fin) bindings as:

$$(|\{x{:}\mathtt{B} \mid p\}|) \doteq \mathtt{True} \qquad (|\{x{:}\mathtt{B}^l \mid p\}|) \doteq p$$

Soundness We prove soundness by showing that the decidable implication $\preceq$-BASE-D approximates the undecidable $\preceq$-BASE.

Theorem 4. *If* $(|\Gamma|) \Rightarrow p_1 \Rightarrow p_2$ *is valid then* $\Gamma \vdash_U \{v{:}B \mid p_1\} \preceq \{v{:}B \mid p_2\}$.

To prove the above, let $VC \doteq (|\Gamma|) \Rightarrow p_1 \Rightarrow p_2$. We prove that if the VC is valid then $\Gamma \vdash_U \{v{:}b \mid p_1\} \preceq \{v{:}b \mid p_2\}$. This fact relies crucially on a notion of *tracking evaluation* which allows us to reduce a closing substitution θ to a lifted substitution $\theta^\perp$, written $\theta \hookrightarrow^*_\perp \theta^\perp$, after which we prove:

Lemma 3. *[Lifting]* $\theta(e) \hookrightarrow^* c$ *iff* $\exists\theta \hookrightarrow^*_\perp \theta^\perp$ *s.t.* $\theta^\perp(e) \hookrightarrow^* c$.

We combine the Lifting Lemma and the equivalence Theorem 2 to prove that the validity of the VC demonstrates the denotational containment $\forall\theta \in [\![\Gamma]\!].[\![\theta(\{v{:}B \mid p_1\})]\!] \subseteq [\![\theta(\{v{:}B \mid p_2\})]\!]$. The soundness of algorithmic typing follows from Theorems 4 and 1:

Theorem 5. *[Soundness of* λ^D *]*

- ***Approximation:*** *If* $\emptyset \vdash_D e : \tau$ *then* $\emptyset \vdash_U e : \tau$.
- ***Crash-Freedom:*** *If* $\emptyset \vdash_D e : \tau$ *then* $e \not\hookrightarrow^* \mathtt{Crash}$.

To prove approximation we need to prove that Lemma 1 holds for each constant, and thus it holds for data constructors. In the metatheory we assume a stronger notion of validity that respects the measure axioms. However, since our implementation does not use axioms and instead, without loss of precision, treats measures as uninterpreted during SMT validity checking, we omit further discussion of axioms for clarity.

2.4 Implementation in LIQUID HASKELL

We have implemented λ^D in LIQUID HASKELL. In § 1.3 we saw real world termination checks. Here claim soundness of LIQUID HASKELL's termination checker, as the checker derives as a the transition from λ^D to Haskell.

2.4.1 Termination

Haskell's recursive functions of type $\texttt{Nat}^{\Downarrow} \rightarrow \tau$ are represented, in GHC's Core [87] as $\texttt{let rec}\ f = \lambda n.e$ that is operationally equivalent to $\texttt{let}\ f = \texttt{tfix}_\tau\ (\lambda n.\lambda f.e)$. Given the type of $\texttt{tfix}_\tau$, checking that f has type $\texttt{Nat}^{\Downarrow} \rightarrow \tau$ reduces to checking e in a *termination-weakened environment* where

$$f{:}\{v{:}\texttt{Nat}^{\Downarrow} \mid v < n\} \rightarrow \tau$$

Thus, LIQUID HASKELL proves termination just as λ^D does: by checking the body in the above environment, where the recursive binder is called with `Nat` inputs that are strictly smaller than n.

Default Metric For example, LIQUID HASKELL proves that

```
fac n = if n == 0 then 1 else n * fac (n-1)
```

has type $\texttt{Nat}^{\Downarrow} \rightarrow \texttt{Nat}^{\Downarrow}$ by typechecking the body of `fac` in a termination-weakened environment $\texttt{fac} : \{\texttt{v}{:}\texttt{Nat}^{\Downarrow} \mid \texttt{v} < \texttt{n}\} \rightarrow \texttt{Nat}^{\Downarrow}$. The recursive call generates the query:

$$\texttt{n}{:}\{0 \leq \texttt{n}\}, \neg(\texttt{n} = 0) \vdash_D \{\texttt{v} = \texttt{n} - 1\} \preceq \{0 \leq \texttt{v} \wedge \texttt{v} < \texttt{n}\}$$

Which reduces to the valid VC:

$$0 \leq \mathtt{n} \wedge \neg(\mathtt{n} = 0) \Rightarrow (\mathtt{v} = \mathtt{n} - 1) \Rightarrow (0 \leq \mathtt{v} \wedge \mathtt{v} < \mathtt{n})$$

proving that `fac` terminates, in essence because the *first parameter* forms a *well-founded decreasing metric*.

Refinements Enable Termination Consider Euclid's GCD:

```
gcd :: a:Nat → {v:Nat | v < a} → Nat
gcd a 0 = a
gcd a b = gcd b (a `mod` b)
```

Here, the first parameter is decreasing, but this requires the fact that the second parameter is smaller than the first and that `mod` returns results smaller than its second parameter. Both facts are easily expressed as refinements, but elude non-extensible checkers [36].

Explicit Termination Metrics The indexed-fixpoint combinator technique is easily extended to cases where some parameter *other* than the first is the well-founded metric. For example, consider:

```
tfac     :: Nat → n:Nat → Nat / [n]
tfac x n | n == 0    = x
         | otherwise = tfac (n*x) (n-1)
```

We specify that the *last parameter* is decreasing by specifying an explicit termination metric `/ [n]` in the type signature. LIQUID HASKELL *desugars* the termination metric into a new `Nat`-valued *ghost parameter* `d` whose value is always equal to the termination metric `n`:

```
tfac :: d:Nat → Nat → {n:Nat | d = n} → Nat
tfac d x n | n == 0    = x
           | otherwise = tfac (n-1) (n*x) (n-1)
```

Type checking, as before, checks the body in an environment where the first argument of `tfac` is weakened, *i.e.*, requires proving `d > n-1`. So, the system needs to know that the ghost argument `d` represents the decreasing metric. We capture this information in the type signature of `tfac` where the *last* argument exactly specifies that `d` is the termination metric `n`, *i.e.*, `d = n`. Note

that since the termination metric can depend on any argument, it is important to refine the last argument, so that all arguments are in scope, with the fact that `d` is the termination metric.

To generalize, desugaring of termination metrics proceeds as follows. Let f be a recursive function with parameters $\overline{x}$ and termination metric $\mu(\overline{x})$. Then LIQUID HASKELL will

- add a `Nat`-valued ghost first parameter d in the definition of f,
- weaken the last argument of f with the refinement $d = \mu(\overline{x})$,
- at each recursive call of $f\ \overline{e}$, apply $\mu(\overline{e})$ as the first argument.

Explicit Termination Expressions Let us now apply the previous technique in a function where none of the parameters themselves decrease across recursive calls, but there is some *expression* that forms the decreasing metric. Consider `range lo hi` (as in § 1.3), which returns the list of `Int`s from `lo` to `hi`: We generalize the explicit metric specification to *expressions* like `hi-lo`. LIQUID HASKELL *desugars* the expression into a new `Nat`-valued *ghost parameter* whose value is always equal to `hi-lo`, that is:

```
range :: lo:Nat → {hi:Nat | hi ≥ lo} → [Nat] / [hi-lo]
range lo hi
  | lo < hi = lo : range (lo + 1) hi
  | _       = []
```

Here, neither parameter is decreasing (indeed, the first one is *increasing*) but `hi-lo` decreases across each call. We generalize the explicit metric specification to *expressions* like `hi-lo`. LIQUID HASKELL *desugars* the expression into a new `Nat`-valued *ghost parameter* whose value is always equal to `hi-lo`, that is:

```
range lo hi = go (hi-lo) lo hi
  where
    go :: d:Nat → lo:Nat → {hi:Nat | d = hi - lo} → [Nat]
    go d lo hi
     | lo < hi = 1 : go (hi-(lo+1)) (lo+1) hi
     | _       = []
```

After which, it proves `go` terminating, by showing that the first argument `d` is a `Nat` that decreases across each recursive call (as in `fac` and `tfac`).

Recursion over Data Types The above strategy generalizes easily to functions that recurse over (finite) data structures like arrays, lists, and trees. In these cases, we simply use *measures* to project the structure onto `Nat`, thereby reducing the verification to the previously seen cases. For each user defined type, *e.g.*

```
data L [sz] a = N | C a (L a)
```

we can define a *measure*

```
measure sz  :: L a → Nat
  sz (C x xs) = 1 + (sz xs)
  sz N        = 0
```

We prove that `map` terminates using the type:

```
map :: (a → b) → xs:L a → L b / [sz xs]
map f (C x xs) = C (f x) (map f xs)
map f N        = N
```

That is, by simply using `(sz xs)` as the decreasing metric.

Generalized Metrics Over Datatypes Finally, in many functions there is no single argument whose (measure) provably decreases. For example, consider:

```
merge :: xs:L a → ys:L a → L a / [sz xs + sz ys]
merge (C x xs) (C y ys)
  | x < y     = x `C` (merge xs  (y `C` ys))
  | otherwise = y `C` (merge (x `C` xs)  ys)
```

from the homonymous sorting routine. Here, neither parameter decreases, but the *sum* of their sizes does. As before LIQUID HASKELL desugars the decreasing expression into a ghost parameter and thereby proves termination (assuming, of course, that the inputs were finite lists, *i.e.* $\mathtt{L}^{\Downarrow}$ *a*).

Automation: Default Size Measures Structural recursion on the first argument is a common pattern in Haskell code. LIQUID HASKELL automates termination proofs for this common case, by allowing users to specify a *size measure* for each data type, (*e.g.* `sz` for `L a`). Now, if *no*

termination metric is given, by default LIQUID HASKELL assumes that the *first* argument whose type has an associated size measure decreases. Thus, in the above, we need not specify metrics for `fac` or `gcd` or `map` as the size measure is automatically used to prove termination. This simple heuristic allows us to automatically prove 67% of recursive functions terminating.

2.4.2 Non-termination

By default, LIQUID HASKELL checks that every function is terminating. We show in §2.5 that this is in fact the overwhelmingly common case in practice. However, annotating a function as `lazy` deactivates LIQUID HASKELL's termination check (and marks the result as a Div type). This allows us to check functions that are non-terminating, and allows LIQUID HASKELL to prove safety properties of programs that manipulate *infinite* data, such as streams, which arise idiomatically with Haskell's lazy semantics. For example, consider the classic `repeat` function:

```
repeat x = x `C` repeat x
```

We cannot use the `tfix` combinators to represent this kind of recursion, and hence, use the non-terminating `fix` combinator instead. I LIQUID HASKELL, we use the `lazy` keyword to denote potentially diverging definitions defined using the non-terminating `fix` combinator.

2.4.3 User Specifications and Type Inference

In program verification it is common that the user provides functional specification that the code should satisfy. In LIQUID HASKELL these specifications can be provided as type signatures for `let`-bound variables. Consider the typechecking rules of Figure 5.3 that is used by λ^D.

$$\frac{\Gamma \vdash e_x : \tau_x \qquad \Gamma, x{:}\tau_x \vdash e : \tau \qquad \Gamma \vdash \tau}{\Gamma \vdash \mathsf{let}\ x = e_x\ \mathsf{in}\ e : \tau}\ \text{T-LET}$$

Note that T-LET *guesses* an appropriate type τ_x for e_x and binds it to x to typecheck e.

LIQUID HASKELL allows the user to specify the type τ_x for top level bindings. For every binding $\mathsf{let}\ x = e_x\ \mathsf{in}\ \ldots$, if the user provides a type specification τ_x, LIQUID HASKELL checks using the appropriate environment (1) that the specified type is well-formed and (2) that the expression e_x typechecks under the specification τ_x. For the other top level bindings, *i.e.* those

without user-provided specifications, as well as all local bindings, LIQUID HASKELL uses the Liquid Types [79] framework to infer refinement types, thus greatly reducing the number of annotations required from the user.

2.5 Evaluation

Our goal is to build a practical and effective SMT & refinement type-based verifier for Haskell. We have shown that lazy evaluation requires the verifier to reason about divergence; we have proposed an approach for implicitly reasoning about divergence by eagerly proving termination, thereby optimizing the precision of the verifier. Next, we describe an experimental evaluation of our approach that uses LIQUID HASKELL to prove termination on the already verified libraries from Chapter 1. Our evaluation seeks to determine whether our approach is *suitable* for a lazy language (*i.e.* do most Haskell functions terminate?), *precise* enough to capture the termination reasons (*i.e.* is LIQUID HASKELL able to prove that most functions terminate?), *usable* without placing an unreasonably high burden on the user in the form of explicit termination annotations, and *effective* enough to enable the verification of functional correctness properties.

Benchmarks As benchmarks, we used the following famous Haskell libraries: `GHC.List` and `Data.List`, which implement many standard list operations, `Data.Set.Splay`, which implements an splay functional set, `Data.Map.Base`, which implements a functional map, `Vector-Algorithms`, which includes a suite of "imperative" array-based sorting algorithms, `Bytestring`, a library for manipulating byte arrays, and `Text`, a library for high-performance Unicode text processing. The verification of functional correctness on our benchmarks is already discussed in § 1.6. Here we focus only on the extra proof obligations required to reason about function termination.

Results Table 1.1 summarizes our experiments, which covered 39 modules totaling 10,209 non-comment lines of source code. The results were collected on a machine with an Intel Xeon X5600 and 32GB of RAM (no benchmark required more than 1GB). Timing data was for runs that performed full verification of safety and functional correctness properties in addition to termination.

Table 2.1. A quantitative evaluation of our experiments. **LOC** is the number of non-comment lines of source code as reported by `sloccount`. **Fun** is the total number of functions in the library. **Rec** is the number of recursive functions. **Div** is the number of functions marked as potentially non-terminating. **Hint** is the number of termination hints, in the form of *termination expressions*, given to LIQUID HASKELL. **Time** is the time, in seconds, required to run LIQUID HASKELL.

Module	LOC	Fun	Rec	Div	Hint	Time
`GHC.List`	309	66	34	5	0	14
`Data.List`	504	97	50	2	6	11
`Data.Map.Base`	1396	180	94	0	12	175
`Data.Set.Splay`	149	35	17	0	7	26
`Bytestring`	3505	569	154	8	73	285
`Vector-Algorithms`	1218	99	31	0	31	85
`Text`	3128	493	124	5	44	481
Total	10209	1539	504	20	173	1080

- *Suitable:* Our approach of eagerly proving termination is in fact, *highly* suitable: of the 504 recursive functions, only 12 functions were *actually* non-terminating (*i.e.* non-inductive). That is, 97.6% of recursive functions are inductively defined.

- *Precise:* Our approach is extremely precise, as refinements provide auxiliary invariants and extensibility that is crucial for proving termination. We successfully *prove* that 96.0% of recursive functions terminate.

- *Usable:* Our approach is highly usable and only places a modest annotation burden on the user. The default metric, namely the first parameter with an associated size measure, suffices to automatically prove 65.7% of recursive functions terminating. Thus, only 34.3% require explicit termination metric, totaling about 1.7 witnesses (about 1 line each) per 100 lines of code.

- *Effective:* Our approach is extremely effective at improving the precision of the overall verifier (by allowing the VC to use facts about binders that provably reduce to values). Without the termination optimization, *i.e.* by only using information for matched-binders (thus in WHNF), LIQUID HASKELL reports 1,395 unique functional correctness warnings – about 1 per 7 lines. With termination information, this number goes to zero.

2.6 Conclusions & Alternative Approaches

Our goal is to use the recent advances in SMT solving to build automated refinement type-based verifiers for Haskell. In this paper, we have made the following advances towards the goal. First, we demonstrated how the classical technique for generating VCs from refinement subtyping queries is unsound under lazy evaluation. Second, we have presented a solution that addresses the unsoundness by stratifying types into those that are inhabited by terms that may diverge, those that must reduce to Haskell values, and those that must reduce to finite values, and have shown how refinement types may themselves be used to soundly verify the stratification. Third, we have developed an implementation of our technique in LIQUID HASKELL and have evaluated the tool on a large corpus comprising 10KLOC of widely used Haskell libraries. Our experiments empirically demonstrate the practical effectiveness of our approach: using refinement types, we were able to prove 96% of recursive functions as terminating, and to crucially use this information to prove a variety of functional correctness properties.

Limitations While our approach is demonstrably effective *in practice*, it relies critically on proving termination, which, while independently useful, is not wholly satisfying *in theory*, as adding divergence shouldn't *break* a safety proof. Our system can prove a program safe, but if the program is modified by making some functions non-deterministically diverge, then, since we rely on termination, we may no longer be able to prove safety. Thus, in future work, it would be valuable to explore *other* ways to reconcile laziness and refinement typing. We outline some routes and the challenging obstacles along them.

A. Convert Lazy To Eager Evaluation One alternative might be to translate the program from lazy to eager evaluation, for example, to replace every (thunk) e with an abstraction $\lambda().e$, and every use of a lazy value x with an application $x\ ()$. After this, we could simply assume eager evaluation, and so the usual refinement type systems could be used to verify Haskell. Alas, no. While sound, this translation doesn't solve the problem of reasoning about divergence. A dependent function type $x{:}\texttt{Int} \rightarrow \{v{:}\texttt{Int} \mid v > x\}$ would be transformed to $x{:}(() \rightarrow \texttt{Int}) \rightarrow \{v{:}\texttt{Int} \mid v > x\ ()\}$. The transformed type is problematic as it uses arbitrary function applications in the refinement logic!

The type is only sensible if $x\ ()$ provably reduces to a value, bringing us back to square one.

B. Explicit Reasoning about Divergence Another alternative is to enrich the refinement logic with a *value predicate* $\mathsf{Val}(x)$ that is true when "x is a value" and use the SMT solver to *explicitly* reason about divergence. (Note that $\mathsf{Val}(x)$ is equivalent to introducing a $\perp$ constant denoting divergence, and writing $(x \neq \perp)$.) Unfortunately, this $\mathsf{Val}(x)$ predicate takes the VCs outside the scope of the standard efficiently decidable logics supported by SMT solvers. To see why, recall the subtyping query from `good`. With explicit value predicates, this subtyping reduces to the VC:

$$(\mathsf{Val}(x) \Rightarrow x \geq 0),\ (\mathsf{Val}(y) \Rightarrow y \geq 0) \Rightarrow (v = y+1) \Rightarrow (v > 0) \tag{2.12}$$

To prove the above valid, we require the knowledge that $(v = y+1)$ implies that y is a value, *i.e.* that $\mathsf{Val}(y)$ holds. This fact, while obvious to a *human* reader, is outside the decidable theories of linear arithmetic of the existing SMT solvers. Thus, existing solvers would be unable to prove (2.12) valid, causing us to reject `good`.

Possible Fix: Explicit Reasoning With Axioms? One possible fix for the above would be to specify a collection of *axioms* that characterize how the value predicate behaves with respect to the other theory operators. For example, we might specify axioms like:

$$\forall x, y, z.(x = y+z) \ \Rightarrow\ (\mathsf{Val}(x) \wedge \mathsf{Val}(y) \wedge \mathsf{Val}(z))$$
$$\forall x, y.(x < y) \ \Rightarrow\ (\mathsf{Val}(x) \wedge \mathsf{Val}(y))$$

etc.. However, this is a non-solution for several reasons. First, it is not clear what a complete set of axioms is. Second, there is the well known loss of predictable checking that arises when using axioms, as one must rely on various brittle, syntactic matching and instantiation heuristics [25]. It is unclear how well these heuristics will work with the sophisticated linear programming-based algorithms used to decide arithmetic theories. Thus, proper support for value predicates could require significant changes to existing decision procedures, making it impossible to use existing SMT solvers.

Possible Fix: Explicit Reasoning With Types? Another possible fix would be to encode the behavior of the value predicates within the refinement types for different operators, after which the predicate itself could be treated as an *uninterpreted function* in the refinement logic [12]. For instance, we could type the primitives:

```
(+) :: x:Int → y:Int → {v | v  =  x + y ∧ Val x ∧ Val y}
(<) :: x:Int → y:Int → {v | v  ⇔  x < y ∧ Val x ∧ Val y}
```

While this approach requires *no* changes to the SMT machinery, it makes specifications complex and verbose. We cannot just add the value predicates to the primitives' specifications. Consider

```
choose b x y = if b then x+1 else y+2
```

To reason about the output of `choose` we must type it as:

```
choose :: Bool → x:Int → y:Int → {v|(v > x ∧ Val x)||(v > y ∧ Val y)}
```

Thus, the value predicates will pervasively clutter all signatures with strictness information, making the system unpleasant to use.

Divergence Requires 3-Valued Logic Finally, for either "fix", the value predicate poses a model-theoretic problem: what is the meaning of $\mathsf{Val}(x)$? One sensible approach is to extend the universe with a family of *distinct* $\perp$ constants, such that $\mathsf{Val}(\perp)$ is false. These constants lead inevitably into a three-valued logic (in order to give meaning to formulas like $\perp = \perp$). Thus, even if we were to find a way to reason with the value predicate via axioms or types, we would have to ensure that we properly handled the 3-valued logic within existing 2-valued SMT solvers.

Future Work Thus, in future work it would be worthwhile to address the above technical and usability problems to enable explicit reasoning with the value predicate. This explicit system would be *more expressive* than our stratified approach, *e.g.* would let us check

```
let x = collatz 10 in 12 `div` x + 1
```

by encoding strictness inside the logic. Nevertheless, we suspect such a verifier would use stratification to eliminate the value predicate in the common case. At any rate, until these hurdles are crossed, we can take comfort in stratified refinement types and can just *eagerly* use termination to prove safety for *lazy* languages.

Acknowledgments The material of this chapter are adapted from the following publication: N. Vazou, E. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, “Refinement Types for Haskell”, ICFP, 2014.

Chapter 3

Abstract Refinement Types

> The purpose of abstraction is not to be vague,
> but to create a new semantic level in which one can be absolutely precise.
>
> *– Edsger W. Dijkstra*

We have presented LIQUID HASKELL, a refinement type checker to Haskell adjusted (using a termination checker) to ensure soundness under Haskell's lazy evaluation. Following standard refinement typing, LIQUID HASKELL reduces refinement type checking to *subtyping* queries of the form $\Gamma \vdash \{\tau : v \mid p\} \preceq \{\tau : v \mid q\}$, where p and q are refinement predicates. These subtyping queries reduce to logical *validity* queries of the form $[\![\Gamma]\!] \wedge p \Rightarrow q$, which can be automatically discharged using SMT solvers [24].

Unfortunately, the automatic verification offered by refinements has come at a price. To ensure decidable checking with SMT solvers, the refinements are quantifier-free predicates drawn from a decidable logic. This significantly limits expressiveness by precluding specifications that enable abstraction over the refinements (*i.e.* invariants). For example, consider the following higher-order for-loop where `set i x v` returns the vector `v` updated at index `i` with the value `x`.

```
for :: Int → Int → a → (Int → a → a) → a
for lo hi x body      = loop lo x
  where loop i x
          | i < hi    = loop (i+1) (body i x)
          | otherwise = x
```

```
initUpto :: Vec a → a → Int → Vec a
initUpto a x n = for 0 n a (\i → set i x)
```

We would like to verify that `initUpto` returns a vector whose *first* `n` elements are equal to `x`. In a first-order setting, we could achieve the above with a loop invariant that asserted that at the $\texttt{i}^{th}$ iteration, the first `i` elements of the vector were already initalized to `x`. However, in our higher-order setting we require a means of *abstracting* over possible invariants, each of which can *depend on* the iteration index `i`. Higher-order logics like Coq and Agda permit such quantification over invariants. Alas, validity in such logics is well outside the realm of decidability, precluding automatic verification.

In this chapter, we present *abstract refinement types* which enable abstraction (quantification) over the refinements of data- and function-types. Our key insight is that we can preserve SMT-based decidable type checking by encoding abstract refinements as *uninterpreted* propositions in the refinement logic. This yields several contributions:

- First, we illustrate how abstract refinements yield a variety of sophisticated means for reasoning about high-level program constructs (§3.1), including: *parametric* refinements for type classes, *index-dependent* refinements for key-value maps, *recursive* refinements for data structures, and *inductive* refinements for higher-order traversal routines.

- Second, we demonstrate that type checking remains decidable (§3.2) by showing a fully automatic procedure that uses SMT solvers, or to be precise, decision procedures based on congruence closure [69] to discharge logical subsumption queries over abstract refinements.

- Third, we show that the crucial problem of *inferring* appropriate instantiations for the (abstract) refinement parameters boils down to inferring (non-abstract) refinement types (§3.2), which we have previously automated via the abstract interpretation framework of Liquid Types [79].

- Finally, we have implemented abstract refinements in LIQUID HASKELL. We present experiments using LIQUID HASKELL to concisely specify and verify a variety of correctness

properties of several programs ranging from microbenchmarks to some widely used libraries (§3.3).

3.1 Overview

We start with a high level overview of abstract refinements, by illustrating how they can be used to uniformly specify and automatically verify various kinds of invariants.

3.1.1 Parametric Invariants

Parametric Invariants via Type Polymorphism Suppose we had a generic comparison `(≤) :: a → a → Bool` as in OCAML. We could use it to write:

```
max     :: a → a → a
max x y = if x ≤ y then y else x

maximum :: [a] → a
maximum (x:xs) = foldr max x xs
```

In essence, the type given for `maximum` states that *for any* `a`, if a list of `a` values is passed into `maximum`, then the returned result is also an `a` value. Hence, for example, if a list of *prime* numbers is passed in, the result is prime, and if a list of *even* numbers is passed in, the result is even. Thus, we can use refinement types [79] to verify

```
type Even = {v:Int | v % 2 = 0 }

maxEvens :: [Int] → Even
maxEvens xs = maximum (0 : xs′)
  where xs′ = [ x | x <- xs, x `mod` 2 == 0]
```

Here the `%` represents the modulus operator in the refinement logic [24] and we type the primitive `mod :: x:Int → y:Int → {v: Int | v = x % y}`. Verification proceeds as follows. Given that `xs :: [Int]`, the system has to verify that `maximum (0 : xs′) :: Even`. To this end, the type parameter of `maximum` is instantiated with the *refined* type `Even`, yielding the instance:

```
maximum :: [Even] → Even
```

Then, `maximum`'s argument should be proved to have type `[Even]`. So, the type parameter of `(:)` is instantiated with `Even`, yielding the instance:

```
(:) :: Even → [Even] → [Even]
```

Finally, the system infers that `0 :: Even` and `xs′ :: [Even]`, *i.e.* the arguments of `(:)` have the expected types, thereby verifying the program. The refinement type instantiations can be inferred, from an appropriate set of logical qualifiers, using the abstract interpretation framework of Liquid Types [79]. Here, once `v%2 = 0` is added to the set of qualifiers, either manually or (as done by our implementation) by automatically scraping predicates from refinements appearing in specification signatures, the refinement type instantiations and hence verification, proceed automatically. Thus, parametric polymorphism offers an easy means of encoding second-order invariants, *i.e.* of quantifying over or parametrizing the invariants of inputs and outputs of functions.

Parametric Invariants via Abstract Refinements Instead, suppose that the comparison operator was monomorphic and only worked for `Int` values. The resulting (monomorphic) signatures

```
max     :: Int → Int → Int
maximum :: [Int] → Int
```

preclude the verification of `maxEvens` (*i.e.* typechecking against the signature shown earlier). This is because the new type of `maximum` merely states that *some* `Int` is returned as output and not necessarily one that enjoys the properties of the values in the input list. This is a shame, since the property clearly still holds. We could type

```
max :: ∀ t ⪯ Int. t → t → t
```

but this route would introduce the complications that surround bounded quantification which could render checking undecidable [75].

To solve this problem, we introduce *abstract refinements* which let us quantify or parameterize a type over its constituent refinements. For example, we can type `max` as

```
max :: ∀ <p :: Int → Bool>. Int<p> → Int<p> → Int<p>
```

where `Int<p>` is an abbreviation for the refinement type `{v:Int | p v}`. Intuitively, an abstract refinement `p` is encoded in the refinement logic as an *uninterpreted function symbol*, which satisfies the *congruence* axiom [69]

$$\forall \overline{X}, \overline{Y} : (\overline{X} = \overline{Y}) \Rightarrow P(\overline{X}) = P(\overline{Y})$$

Thus, it is trivial to verify, with an SMT solver, that `max` enjoys the above type: the input types ensure that both `p x` and `p y` hold and hence the returned value in either branch satisfies the refinement `{v:Int | p v}`, thereby ensuring the output type. By the same reasoning, we can generalize the type of `maximum` to

```
maximum :: ∀ <p :: Int → Bool>. [Int<p>] → Int<p>
```

Consequently, we can recover the verification of `maxEvens`. Now, instead of instantiating a *type* parameter, we simply instantiate the *refinement* parameter of `maximum` with the concrete refinement `{\v → v % 2 = 0}`, after which type checking proceeds as usual [79]. Later, we show how to retain automatic verification by inferring refinement parameter instantiations via liquid typing (§ 3.2.4).

Parametric Invariants and Type Classes The example above regularly arises in practice, due to type classes. In Haskell, the functions above are typed

```
(≤)     :: (Ord a) ⇒ a → a → Bool
max     :: (Ord a) ⇒ a → a → a
maximum :: (Ord a) ⇒ [a] → a
```

We might be tempted to ignore the typeclass constraint and treat `maximum` as `[a] → a`. This would be quite unsound, as typeclass predicates preclude universal quantification over refinement types. Consider the function `sum :: (Num a) ⇒ [a] → a` which adds the elements of a list. The `Num` class constraint implies that numeric operations occur in the function, so if we pass `sum` a list of odd numbers, we are *not* guaranteed to get back an odd number.

Thus, how do we soundly verify the desired type of `maxEvens` without instantiating class predicated type parameters with arbitrary refinement types? First, via the same analysis as the

monomorphic `Int` case, we establish that

```
max     :: ∀ <p :: a → Bool>. (Ord a) ⇒ a<p> → a<p> → a<p>
maximum:: ∀ <p :: a → Bool>. (Ord a) ⇒ [a<p>] → a<p>
```

Next, at the call-site for `maximum` in `maxEvens` we instantiate the type variable `a` with `Int` and the abstract refinement `p` with `{\v → v % 2 = 0}` after which, the verification proceeds as described earlier (for the `Int` case). Thus, abstract refinements allow us to quantify over invariants without relying on parametric polymorphism, even in the presence of type classes.

3.1.2 Index-Dependent Invariants

Next, we illustrate how abstract invariants allow us to specify and verify index-dependent invariants of key-value maps. To this end, we develop a small library of *extensible vectors* encoded, for purposes of illustration, as functions from `Int` to some generic range `a`. Formally, we specify vectors as

```
data Vec a <dom :: Int → Bool, rng :: Int → a → Bool>
  = V (i:Int<dom> → a <rng i>)
```

Here, we are parameterizing the definition of the type `Vec` with *two* abstract refinements, `dom` and `rng`, which respectively describe the *domain* and *range* of the vector. That is, `dom` describes the set of *valid* indices and `r` specifies an invariant relating each `Int` index with the value stored at that index.

Creating Vectors We can use the following basic functions to create vectors:

```
empty :: ∀ <p :: Int → a → Bool>.Vec<{\_ → False}, p> a
empty = V (\_ → error "Empty Vec")


create :: x:a → Vec <{\_ → True}, {\_ v → v = x}> a
create x = V (\_ → x)
```

The signature for `empty` states that its domain is empty (*i.e.* is the set of indices satisfying the predicate `False`) and that the range satisfies `any` invariant. The signature for `create`, instead, defines a *constant* vector that maps every index to the constant `x`.

Accessing Vectors We can write the following `get` function for reading the contents of a vector at a given index:

```
get :: ∀ <d :: Int → Bool, r :: Int → a → Bool>
       i:Int<d> → Vec<d, r> a → a<r i>
get i (V f) = f i
```

The signature states that for any domain `d` and range `r`, if the index `i` is a valid index, *i.e.* is of type, `Int<d>` then the returned value is an `a` that additionally satisfies the range refinement at the index `i`. The type for `set`, which *updates* the vector at a given index, is even more interesting, as it allows us to *extend* the domain of the vector:

```
set :: ∀ <d :: Int → Bool, r :: Int → a → Bool>
       i:Int<d>
    → a<r i>
    → Vec<d ∧ {\k → k ≠ i}, r> a
    → Vec<d, r> a
set i v (V f) = V (\k → if k == i then v else f k)
```

The signature for `set` requires that (a) the input vector is defined everywhere at `d` *except* the index `i` and (b) the value supplied must be of type `a<r i>`, *i.e.* satisfy the range relation at the index `i` at which the vector is being updated. The signature ensures that the output vector is defined at `d` and each value satisfies the index-dependent range refinement `r`. Note that it is legal to call `get` with a vector that is *also* defined at the index `i` since, by contravariance, such a vector is a subtype of that required by (a).

Initializing Vectors Next, we can write the following function, `init`, that "loops" over a vector, to `set` each index to a value given by some function.

```
initialize :: ∀ <r :: Int → a → Bool>.
              (z: Int → a<r z>)
           → i: {v: Int | v ≥ 0}
           → n: Int
           → Vec <{\v → 0 ≤ v ∧ v < i}, r> a
           → Vec <{\v → 0 ≤ v ∧ v < n}, r> a
```

```
initialize f i n a
  | i ≥ n    = a
  | otherwise = initialize f (i+1) n (set i (f i) a)
```

The signature requires that (a) the higher-order function `f` produces values that satisfy the range refinement `r` and (b) the vector is initialized from `0` to `i`. The function ensures that the output vector is initialized from `0` through `n`. We can thus verify that

```
idVec   :: Vec <{\v → 0 ≤ v ∧ v<n}, {\i v → v=i}> Int
idVec n = initialize (\i → i) 0 n empty
```

i.e. `idVec` returns a vector of size `n` where each key is mapped to itself. Thus, abstract refinement types allow us to verify low-level idioms such as the incremental initialization of vectors, which have previously required special analyses [37, 44, 22].

Null-Terminated Strings We can also use abstract refinements to verify code which manipulates C-style null-terminated strings, represented as `Char` vectors for ease of exposition. Formally, a null-terminated string of size `n` has the type

```
type NullTerm n
  = Vec <{\v → 0 ≤ v<n}, {\i v → i=n-1 ⇒ v='\0'}> Char
```

The above type describes a length-`n` vector of characters whose last element must be a null character, signalling the end of the string. We can use this type in the specification of a function, `upperCase`, which iterates through the characters of a string, uppercasing each one until it encounters the null terminator:

```
upperCase :: n:{v: Int| v>0} → NullTerm n → NullTerm n
upperCase n s = ucs 0 s
  where
    ucs i s = case get i s of
                '\0' → s
                c    → ucs (i + 1) (set i (toUpper c) s)
```

Note that the length parameter `n` is provided solely as a "witness" for the length of the string `s`, which allows us to use the length of `s` in the type of `upperCase`; `n` is not used in the computation. In order to establish that each call to `get` accesses string `s` within its bounds, our type system

must establish that, at each call to the inner function `ucs`, `i` satisfies the type `{v: Int | 0 ≤ v ∧ v < n}`. This invariant is established as follows. First, the invariant trivially holds on the first call to `ucs`, as `n` is positive and `i` is 0. Second, we assume that `i` satisfies the type `{v: Int | 0 ≤ v ∧ v < n}`, and, further, we know from the types of `s` and `get` that `c` has the type `{v: Char | i = n - 1 ⇒ v = '\0'}`. Thus, if `c` is non-null, then `i` cannot be equal to `n - 1`. This allows us to strengthen our type for `i` in the else branch to `{v: Int | 0 ≤ v ∧ v < n - 1}` and thus to conclude that the value `i + 1` recursively passed as the `i` parameter to `ucs` satisfies the type `{v: Int | 0 ≤ v ∧ v < n}`, establishing the inductive invariant and thus the safety of the `upperCase` function.

Memoization Next, let us illustrate how the same expressive signatures allow us to verify memoizing functions. We can specify to the SMT solver the definition of the Fibonacci function via an uninterpreted function `fib` and an axiom:

```
measure fib :: Int → Int
axiom: ∀ i. (fib i) = if i ≤ 1 then 1 else fib (i-1) + fib (i-2)
```

Next, we define a type alias `FibV` for the vector whose values are either `0` (*i.e.* undefined) or equal to the Fibonacci number of the corresponding index.

```
type FibV = Vec<{\_→True},{\i v→ v ≠ 0 ⇒ v = fib i}> Int
```

Finally, we can use the above alias to verify `fastFib`, an implementation of the Fibonacci function, which uses a vector memoize intermediate results

```
fastFib   :: n:Int → {v:Int | v = fib(n)}
fastFib n = snd $ fibMemo (create 0) n

fibMemo :: FibV → i:Int → (FibV, {v: Int | v = fib(i)})
fibMemo t i
  | i ≤ 1    = (t, 1)
  | otherwise = case get i t of
                  0 → let (t1, n1) = fibMemo t  (i-1)
                          (t2, n2) = fibMemo t1 (i-2)
                          n        = n1 + n2
                      in  (set i n t2,  n)
```

```
n → (t, n)
```

Thus, abstract refinements allow us to define key-value maps with index-dependent refinements for the domain and range. Quantification over the domain and range refinements allows us to define generic access operations (*e.g.* `get`, `set`, `create`, `empty`) whose types enable us establish a variety of precise invariants.

3.1.3 Recursive Invariants

Next, we turn our attention to recursively defined datatypes and show how abstract refinements allow us to specify and verify high-level invariants that relate the elements of a recursive structure. Consider the following refined definition for lists:

```
data [a] <p :: a → a → Bool> where
  []  :: [a]<p>
  (:) :: h:a → [a<p h>]<p> → [a]<p>
```

The definition states that a value of type `[a]<p>` is either empty (`[]`) or constructed from a pair of a *head* `h::a` and a *tail* of a list of `a` values *each* of which satisfies the refinement `(p h)`. Furthermore, the abstract refinement `p` holds recursively within the tail, ensuring that the relationship `p` holds between *all* pairs of list elements.

Thus, by plugging in appropriate concrete refinements, we can define the following aliases, which correspond to the informal notions implied by their names:

```
type IncrList a = [a]<{\h v → h ≤ v}>
type DecrList a = [a]<{\h v → h ≥ v}>
type UniqList a = [a]<{\h v → h ≠ v}>
```

That is, `IncrList a` (resp. `DecrList a`) describes a list sorted in increasing (resp. decreasing) order and `UniqList a` describes a list of *distinct* elements, *i.e.* not containing any duplicates. We can use the above definitions to verify

```
[1, 2, 3, 4] :: IncrList Int
[4, 3, 2, 1] :: DecrList Int
[4, 1, 3, 2] :: UniqList Int
```

More interestingly, we can verify that the usual algorithms produce sorted lists:

```
insertSort :: (Ord a) ⇒ [a] → IncrList a
insertSort []     = []
insertSort (x:xs) = insert x (insertSort xs)

insert :: (Ord a) ⇒ a → IncrList a → IncrList a
insert y []       = [y]
insert y (x:xs)
  | y ≤ x         = y : x : xs
  | otherwise     = x : insert y xs
```

Thus, abstract refinements allow us to *decouple* the definition of the list from the actual invariants that hold. This, in turn, allows us to conveniently reuse the same underlying (non-refined) type to implement various algorithms unlike, say, singleton-type based implementations which require up to three different types of lists (with three different "nil" and "cons" constructors [84]). This, makes abstract refinements convenient for verifying complex sorting implementations like that of `Data.List.sort` which, for efficiency, use lists with different properties (*e.g.* increasing and decreasing).

Multiple Recursive Refinements We can define recursive types with multiple parameters. For example, consider the following refined version of a type used to encode functional maps (`Data.Map`):

```
data Tree k v <l :: k → k → Bool, r :: k → k → Bool>
  = Bin { key   :: k
        , value :: v
        , left  :: Tree <l, r> (k <l key>) v
        , right :: Tree <l, r> (k <r key>) v }
  | Tip
```

The abstract refinements `l` and `r` relate each `key` of the tree with *all* the keys in the *left* and *right* subtrees of `key`, as those keys are respectively of type `k <l key>` and `k <r key>`. Thus, if we instantiate the refinements with the following predicates

```
type BST k v     = Tree<{\x y → x> y},{\x y→ x< y}> k v
type MinHeap k v = Tree<{\x y → x≤y},{\x y→ x≤y}> k v
```

```
type MaxHeap k v = Tree<{\x y → x ≥ y},{\x y→ x ≥ y}> k v
```

then `BST k v`, `MinHeap k v` and `MaxHeap k v` denote exactly binary-search-ordered, min-heap-ordered, and max-heap-ordered trees (with keys and values of types `k` and `v`). We demonstrate in (§ 3.3) how we use the above types to automatically verify ordering properties of complex, full-fledged libraries.

3.1.4 Inductive Invariants

Finally, we explain how abstract refinements allow us to formalize some kinds of structural induction within the type system.

Measures First, let us formalize a notion of *length* for lists within the refinement logic. To do so, we define a special `len` measure by structural induction

```
measure len :: [a] → Int
  len []       = 0
  len (x:xs)   = 1 + len(xs)
```

We use the measures to automatically strengthen the types of the data constructors 2.1.6:

```
data [a] where
  []  :: {v:[a] | len v = 0}
  (:) :: a → xs:[a] → {v:[a]| len v = 1 + len xs}
```

Note that the symbol `len` is encoded as an *uninterpreted* function in the refinement logic, and is, except for the congruence axiom, opaque to the SMT solver. The measures are guaranteed, by construction, to terminate and so we can soundly use them as uninterpreted functions in the refinement logic. Notice also, that we can define *multiple* measures for a type; in this case we simply conjoin the refinements from each measure when refining each data constructor.

With these strengthened constructor types, we can verify, for example, that `append` produces a list whose length is the sum of the input lists' lengths:

```
append :: l:[a] → m:[a] → {v:[a]| len v = len l + len m}
append []     zs = zs
append (y:ys) zs = y : append ys zs
```

However, consider an alternate definition of `append` that uses `foldr`

```
append ys zs = foldr (:) zs ys
```

where `foldr :: (a → b → b) → b → [a] → b`. It is unclear how to give `foldr` a (first-order) refinement type that captures the rather complex fact that the fold-function is "applied" all over the list argument, or, that it is a catamorphism. Hence, hitherto, it has not been possible to verify the second definition of `append`.

Typing Folds Abstract refinements allow us to solve this problem with a very expressive type for `foldr` whilst remaining firmly within the boundaries of SMT-based decidability. We write a slightly modified fold:

```
foldr :: ∀ <p :: [a] → b → Bool>.
          (xs:[a] → x:a → b <p xs> → <p (x:xs)>)
       → b<p []>
       → ys:[a]
       → b<p ys>
foldr op b []     = b
foldr op b (x:xs) = op xs x (foldr op b xs)
```

The trick is simply to quantify over the relationship `p` that `foldr` establishes between the input list `xs` and the output `b` value. This is formalized by the type signature, which encodes an induction principle for lists: the base value `b` must (1) satisfy the relation with the empty list, and the function `op` must take (2) a value that satisfies the relationship with the tail `xs` (we have added the `xs` as an extra "ghost" parameter to `op`), (3) a head value `x`, and return (4) a new folded value that satisfies the relationship with `x:xs`. If all the above are met, then the value returned by `foldr` satisfies the relation with the input list `ys`. This scheme is not novel in itself [8] — what is new is the encoding, via uninterpreted predicate symbols, in an SMT-decidable refinement type system.

Using Folds Finally, we can use the expressive type for the above `foldr` to verify various inductive properties of client functions:

```
length :: zs:[a] → {v: Int | v = len zs}
length = foldr (\_ _ n → n + 1) 0


append :: l:[a] → m:[a] → {v:[a]| len v = len l + len m}
```

```
append ys zs = foldr (\_ → (:)) zs ys
```

The verification proceeds by just (automatically) instantiating the refinement parameter `p` of `foldr` with the concrete refinements, via Liquid typing:

```
{\xs v → v = len xs}                 -- for length
{\xs v → len v = len xs + len zs}    -- for append
```

3.2 Syntax and Semantics

Next, we present a core calculus λ_P that formalizes the notion of abstract refinements. We start with the syntax (§ 3.2.1), present the typing rules (§ 3.2.2), show soundness via a reduction to contract calculi [51, 6] (§ 4.2.4), and inference via Liquid types (§ 3.2.4).

3.2.1 Syntax

Figure 3.1 summarizes the syntax of our core calculus λ_P which is a polymorphic λ-calculus extended with abstract refinements. We write b, $\{v : b \mid e\}$ and $b\langle p \rangle$ to abbreviate $\{v : b\langle \mathit{True} \rangle \mid \mathit{True}\}$, $\{v : b\langle \mathit{True} \rangle \mid e\}$, and $\{v : b\langle p \rangle \mid \mathit{True}\}$ respectively. We say a type or schema is *non-refined* if all the refinements in it are *True*. We write $\overline{z}$ to abbreviate a sequence $z_1 \ldots z_n$.

Expressions λ_P expressions include the standard variables x, primitive constants c, λ-abstraction $\lambda x : \tau.e$, application $e\ e$, type abstraction $\Lambda\alpha.e$, and type application $e\,[\tau]$. The parameter τ in the type application is a *refinement type*, as described shortly. The two new additions to λ_P are the refinement abstraction $\Lambda\pi : \tau.e$, which introduces a refinement variable π (together with its type τ), which can appear in refinements inside e, and the corresponding refinement application $e\,[e]$.

Refinements A *concrete refinement* e is a boolean valued expression e drawn from a strict subset of the language of expressions which includes only terms that (a) neither diverge nor crash and (b) can be embedded into an SMT decidable refinement logic including the theory of linear arithmetic and uninterpreted functions. An *abstract refinement* p is a conjunction of refinement variable applications of the form $\pi\ \overline{e}$.

Types and Schemas The basic types of λ_P include the base types `Int` and `Bool` and type variables α. An *abstract refinement type* τ is either a basic type refined with an abstract and concrete

$$
\begin{array}{rrcl}
\textbf{\textit{Expressions}} & e & ::= & x \mid c \mid \lambda x:\tau.e \mid e\ e \\
 & & \mid & \Lambda\alpha.e \mid e[\tau] \mid \Lambda\pi:\tau.e \mid e[e] \\
\textbf{\textit{Abstract Refinements}} & p & ::= & \mathit{True} \mid p \wedge \pi\ \overline{e} \\
\textbf{\textit{Basic Types}} & b & ::= & \texttt{Int} \mid \texttt{Bool} \mid \alpha \\
\textbf{\textit{Abstract Refinement Types}} & \tau & ::= & \{v : b\langle p\rangle \mid e\} \mid \{v : (x:\tau) \to \tau \mid e\} \\
\textbf{\textit{Abstract Refinement Schemas}} & \sigma & ::= & \tau \mid \forall\alpha.\sigma \mid \forall\pi:\tau.\sigma
\end{array}
$$

Figure 3.1. Syntax of Expressions, Refinements, Types and Schemas of λ_P.

refinements, $\{v : b\langle p\rangle \mid e\}$, or a dependent function type where the parameter x can appear in the refinements of the output type. We include refinements for functions, as refined type variables can be replaced by function types. However, typechecking ensures these refinements are trivially true. Finally, types can be quantified over refinement variables and type variables to yield abstract refinement schemas.

3.2.2 Static Semantics

Next, we describe the static semantics of λ_P by describing the typing judgments and derivation rules. Most of the rules are standard [72, 79, 51, 7]; we discuss only those pertaining to abstract refinements.

Judgments A type environment Γ is a sequence of type bindings $x : \sigma$. We use environments to define three kinds of typing judgments.

Wellformedness judgments ($\Gamma \vdash \sigma$) state that a type schema σ is well-formed under environment Γ, that is, the refinements in σ are boolean expressions in the environment Γ. The wellformedness rules check that the concrete and abstract refinements are indeed `Bool`-valued expressions in the appropriate environment. The key rule is WF-BASE, which checks, as usual, that the (concrete) refinement e is boolean and additionally, that the abstract refinement p applied to the value v is also boolean. This latter fact is established by WF-RAPP which checks that each refinement variable application $\pi\ \overline{e}\ v$ is also of type `Bool` in the given environment.

Subtyping judgments ($\Gamma \vdash \sigma_1 \preceq \sigma_2$) state that the type schema σ_1 is a subtype of the type schema σ_2 under environment Γ, that is, when the free variables of σ_1 and σ_2 are bound to values

Well-Formedness $\boxed{\Gamma \vdash \sigma}$

$$\frac{}{\Gamma \vdash \mathit{True}(v)}\ \textsc{WF-True} \qquad \frac{\Gamma \vdash p(v) \quad \Gamma \vdash \pi\ \overline{e}\ v : \texttt{Bool}}{\Gamma \vdash (p \wedge \pi\ \overline{e})(v)}\ \textsc{WF-RApp}$$

$$\frac{\Gamma, v : b \vdash e : \texttt{Bool} \quad \Gamma, v : b \vdash p(v) : \texttt{Bool}}{\Gamma \vdash \{v : b\langle p \rangle \mid e\}}\ \textsc{WF-Base}$$

$$\frac{\Gamma \vdash e : \texttt{Bool} \quad \Gamma \vdash \tau_x \quad \Gamma, x : \tau_x \vdash \tau}{\Gamma \vdash \{v : (x : \tau_x) \to \tau \mid e\}}\ \textsc{WF-Fun}$$

$$\frac{\Gamma, \pi : \tau \vdash \sigma}{\Gamma \vdash \forall \pi : \tau . \sigma}\ \textsc{WF-Abs-}\pi \qquad \frac{\Gamma, \alpha \vdash \sigma}{\Gamma \vdash \forall \alpha . \sigma}\ \textsc{WF-Abs-}\alpha$$

Subtyping $\boxed{\Gamma \vdash \sigma_1 \preceq \sigma_2}$

$$\frac{\text{SMT-Valid}([\![\Gamma]\!] \wedge [\![p_1\ v]\!] \wedge [\![e_1]\!] \Rightarrow [\![p_2\ v]\!] \wedge [\![e_2]\!])}{\Gamma \vdash \{v : b\langle p_1 \rangle \mid e_1\} \preceq \{v : b\langle p_2 \rangle \mid e_2\}}\ \preceq\textsc{-Base}$$

$$\frac{\Gamma \vdash \tau_2 \preceq \tau_1 \quad \Gamma, x_2 : \tau_2 \vdash \tau_1'\,[x_1 \mapsto x_2] \preceq \tau_2'}{\Gamma \vdash \{v : (x_1 : \tau_1) \to \tau_1' \mid e_1\} \preceq \{v : (x_2 : \tau_2) \to \tau_2' \mid \mathit{True}\}}\ \preceq\textsc{-Fun}$$

$$\frac{\Gamma, \pi : \tau \vdash \sigma_1 \preceq \sigma_2}{\Gamma \vdash \forall \pi : \tau . \sigma_1 \preceq \forall \pi : \tau . \sigma_2}\ \preceq\textsc{-RVar} \qquad \frac{\Gamma \vdash \sigma_1 \preceq \sigma_2}{\Gamma \vdash \forall \alpha . \sigma_1 \preceq \forall \alpha . \sigma_2}\ \preceq\textsc{-Poly}$$

Type Checking $\boxed{\Gamma \vdash e : \sigma}$

$$\frac{\Gamma \vdash e : \sigma_2 \quad \Gamma \vdash \sigma_2 \preceq \sigma_1 \quad \Gamma \vdash \sigma_1}{\Gamma \vdash e : \sigma_1}\ \textsc{T-Sub} \qquad \frac{}{\Gamma \vdash c : tc(c)}\ \textsc{T-Const}$$

$$\frac{x : \{v : b\langle p \rangle \mid e\} \in \Gamma}{\Gamma \vdash x : \{v : b\langle p \rangle \mid e \wedge v = x\}}\ \textsc{T-Var-Base} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{T-Var}$$

$$\frac{\Gamma, x : \tau_x \vdash e : \tau \quad \Gamma \vdash \tau_x}{\Gamma \vdash \lambda x : \tau_x . e : (x : \tau_x) \to \tau}\ \textsc{T-Fun} \qquad \frac{\Gamma \vdash e_1 : (x : \tau_x) \to \tau \quad \Gamma \vdash e_2 : \tau_x}{\Gamma \vdash e_1\ e_2 : \tau\,[x \mapsto e_2]}\ \textsc{T-App}$$

$$\frac{\Gamma, \alpha \vdash e : \sigma}{\Gamma \vdash \Lambda \alpha . e : \forall \alpha . \sigma}\ \textsc{T-Gen} \qquad \frac{\Gamma \vdash e : \forall \alpha . \sigma \quad \Gamma \vdash \tau}{\Gamma \vdash e\,[\tau] : \sigma\,[\alpha \mapsto \tau]}\ \textsc{T-Inst}$$

$$\frac{\Gamma, \pi : \tau \vdash e : \sigma \quad \Gamma \vdash \tau}{\Gamma \vdash \Lambda \pi : \tau . e : \forall \pi : \tau . \sigma}\ \textsc{T-PGen} \qquad \frac{\Gamma \vdash e : \forall \pi : \tau . \sigma \quad \Gamma \vdash \lambda \overline{x : \tau_x} . e' : \tau}{\Gamma \vdash e\,[\lambda \overline{x : \tau_x} . e'] : \sigma[\pi \rhd \lambda \overline{x : \tau_x} . e']}\ \textsc{T-PInst}$$

Figure 3.2. Well-formedness, Subtyping and Type Checking of λ_P.

described by Γ, the set of values described by σ_1 is contained in the set of values described by σ_2. The rules are standard except for $\preceq$-VAR, which encodes the base types' abstract refinements p_1 and p_2 with conjunctions of *uninterpreted predicates* $[\![p_1\ v]\!]$ and $[\![p_2\ v]\!]$ in the refinement logic as follows:

$$\begin{aligned} [\![\mathit{True}\ v]\!] &\doteq \mathit{True} \\ [\![(p \wedge \pi\ \overline{e})\ v]\!] &\doteq [\![p\ v]\!] \wedge \pi([\![e_1]\!], \ldots, [\![e_n]\!], v) \end{aligned}$$

where $\pi(\overline{e})$ is a term in the refinement logic corresponding to the application of the uninterpreted predicate symbol π to the arguments $\overline{e}$.

Typing judgments ($\Gamma \vdash e : \sigma$) state that the expression e has the type schema σ under environment Γ, that is, when the free variables in e are bound to values described by Γ, the expression e will evaluate to a value described by σ. The type checking rules are standard except for T-PGEN and T-PINST, which pertain to abstraction and instantiation of abstract refinements. The rule T-PGEN is the same as T-FUN: we simply check the body e in the environment extended with a binding for the refinement variable π. The rule T-PINST checks that the concrete refinement is of the appropriate (unrefined) type τ, and then replaces all (abstract) applications of π inside σ with the appropriate (concrete) refinement e' with the parameters $\overline{x}$ replaced with arguments at that application. Formally, this is represented as $\sigma[\pi \rhd \lambda \overline{x:\tau}.e']$ which is σ with each base type transformed as

$$\begin{aligned} \{v : b\langle p \rangle \mid e\}[\pi \rhd z] &\doteq \left\{v : b\langle p'' \rangle \mid e \wedge e''\right\} \\ \text{where} \quad (p'', e'') &\doteq \mathsf{Apply}(p, \pi, z, \mathit{True}, \mathit{True}) \end{aligned}$$

Apply replaces each application of π in p with the corresponding conjunct in e'', as

$$\begin{aligned}
\mathsf{Apply}(\mathit{True}, \cdot, \cdot, p', e') &\doteq (p', e') \\
\mathsf{Apply}(p \wedge \pi'\ \overline{e}, \pi, z, p', e') &\doteq \mathsf{Apply}(p, \pi, z, p' \wedge \pi'\ \overline{e}, e') \\
\mathsf{Apply}(p \wedge \pi\ \overline{e}, \pi, \lambda\overline{x:\tau}.e'', p', e') &\doteq \mathsf{Apply}(p, \pi, \lambda\overline{x:\tau}.e'', p', e' \wedge e''\,[\overline{x} \mapsto \overline{e}, v])
\end{aligned}$$

In other words, the instantiation can be viewed as two symbolic reduction steps: first replacing the refinement variable with the concrete refinement, and then "beta-reducing" concrete refinement with the refinement variable's arguments. For example,

$$\{v : \mathtt{Int}\langle \pi\ y\rangle \mid v > 10\}[\pi \rhd \lambda x_1 : \tau_1.\lambda x_2 : \tau_2.x_1 < x_2] \doteq \{v : \mathtt{Int} \mid v > 10 \wedge y < v\}$$

3.2.3 Soundness

As hinted by the discussion about refinement variable instantiation, we can intuitively think of abstract refinement variables as *ghost* program variables whose values are boolean-valued functions. Hence, abstract refinements are a special case of higher-order contracts, that can be statically verified using uninterpreted functions. (Since we focus on static checking, we don't care about the issue of blame.) We formalize this notion by translating λ_P programs into the contract calculus $\mathrm{F_H}$ of [6] and use this translation to define the dynamic semantics and establish soundness.

Translation We translate λ_P schemes σ to $\mathrm{F_H}$ schemes $\langle\!|\sigma|\!\rangle$ as by translating abstract refinements into contracts, and refinement abstraction into function types:

$$\begin{aligned}
\langle\!|\mathit{True}\ v|\!\rangle &\doteq \mathit{True} & \langle\!|\forall\pi : \tau.\sigma|\!\rangle &\doteq (\pi : \langle\!|\tau|\!\rangle) \to \langle\!|\sigma|\!\rangle \\
\langle\!|(p \wedge \pi\ \overline{e})\ v|\!\rangle &\doteq \langle\!|p\ v|\!\rangle \wedge \pi\ \overline{e}\ v & \langle\!|\forall\alpha.\sigma|\!\rangle &\doteq \forall\alpha.\langle\!|\sigma|\!\rangle \\
\langle\!|\,\{v : b\langle p\rangle \mid e\}\,|\!\rangle &\doteq \{v : b \mid e \wedge \langle\!|p\ v|\!\rangle\} & \langle\!|(x : \tau_1) \to \tau_2|\!\rangle &\doteq (x : \langle\!|\tau_1|\!\rangle) \to \langle\!|\tau_2|\!\rangle
\end{aligned}$$

Similarly, we translate λ_P terms e to $\mathrm{F_H}$ terms $\langle\!|e|\!\rangle$ by converting refinement abstraction and

application to λ-abstraction and application

$$\begin{array}{rlrl}
\langle\!|x|\!\rangle & \doteq x & \langle\!|c|\!\rangle & \doteq c \\
\langle\!|\lambda x:\tau.e|\!\rangle & \doteq \lambda x:\langle\!|\tau|\!\rangle.\langle\!|e|\!\rangle & \langle\!|e_1\ e_2|\!\rangle & \doteq \langle\!|e_1|\!\rangle\ \langle\!|e_2|\!\rangle \\
\langle\!|\Lambda\alpha.e|\!\rangle & \doteq \Lambda\alpha.\langle\!|e|\!\rangle & \langle\!|e\,[\tau]\,|\!\rangle & \doteq \langle\!|e|\!\rangle\ \langle\!|\tau|\!\rangle \\
\langle\!|\Lambda\pi:\tau.e|\!\rangle & \doteq \lambda\pi:\langle\!|\tau|\!\rangle.\langle\!|e|\!\rangle & \langle\!|e_1\,[e_2]\,|\!\rangle & \doteq \langle\!|e_1|\!\rangle\ \langle\!|e_2|\!\rangle
\end{array}$$

Translation Properties We can show by induction on the derivations that the type derivation rules of λ_P *conservatively approximate* those of $\mathrm{F_H}$. Formally,

- If $\Gamma \vdash \tau$ then $\langle\!|\Gamma|\!\rangle \vdash_H \langle\!|\tau|\!\rangle$,
- If $\Gamma \vdash \tau_1 \preceq \tau_2$ then $\langle\!|\Gamma|\!\rangle \vdash_H \langle\!|\tau_1|\!\rangle <: \langle\!|\tau_2|\!\rangle$,
- If $\Gamma \vdash e : \tau$ then $\langle\!|\Gamma|\!\rangle \vdash_H \langle\!|e|\!\rangle : \langle\!|\tau|\!\rangle$.

Soundness Thus rather than re-prove preservation and progress for λ_P, we simply use the fact that the type derivations are conservative to derive the following preservation and progress corollaries from [6]:

- **Preservation:** If $\emptyset \vdash e : \tau$ and $\langle\!|e|\!\rangle \longrightarrow e'$ then $\emptyset \vdash_H e' : \langle\!|\tau|\!\rangle$
- **Progress:** If $\emptyset \vdash e : \tau$, then either $\langle\!|e|\!\rangle \longrightarrow e'$ or $\langle\!|e|\!\rangle$ is a value.

Note that, in a contract calculus like $\mathrm{F_H}$, subsumption is encoded as a *upcast*. However, if subtyping relation can be statically guaranteed (as is done by our conservative SMT based subtyping) then the upcast is equivalent to the identity function and can be eliminated. Hence, $\mathrm{F_H}$ terms $\langle\!|e|\!\rangle$ translated from well-typed λ_P terms e have no casts.

3.2.4 Refinement Inference

Our design of abstract refinements makes it particularly easy to perform type inference via Liquid typing, which is crucial for making the system usable by eliminating the tedium of instantiating refinement parameters all over the code. (With value-dependent refinements,

one cannot simply use, say, unification to determine the appropriate instantations, as is done for classical type systems). We briefly recall how Liquid types work, and sketch how they are extended to infer refinement instantiations.

Liquid Types The Liquid Types method infers refinements in three steps. First, we create refinement *templates* for the unknown, to-be-inferred refinement types. The *shape* of the template is determined by the underlying (non-refined) type it corresponds to, which can be determined from the language's underlying (non-refined) type system. The template is just the shape refined with fresh refinement variables κ denoting the unknown refinements at each type position. For example, from a type $(x : \texttt{Int}) \rightarrow \texttt{Int}$ we create the template $(x : \{v : \texttt{Int} \mid \kappa_x\}) \rightarrow \{v : \texttt{Int} \mid \kappa\}$. Second, we perform type checking using the templates (in place of the unknown types). Each wellformedness check becomes a wellformedness constraint over the templates, and hence over the individual κ, constraining which variables can appear in κ. Each subsumption check becomes a subtyping constraint between the templates, which can be further simplified, via syntactic subtyping rules, to a logical implication query between the variables κ. Third, we solve the resulting system of logical implication constraints (which can be cyclic) via abstract interpretation — in particular, monomial predicate abstraction over a set of logical qualifiers [33, 79]. The solution is a map from κ to conjunctions of qualifiers, which, when plugged back into the templates, yields the inferred refinement types.

Inferring Refinement Instantiations The key to making abstract refinements practical is a means of synthesizing the appropriate arguments e' for each refinement application $e\,[e']$. Note that for such applications, we can, from e, determine the non-refined type of e', which is of the form $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \texttt{Bool}$. Thus, e' has the template $\lambda x_1 : \tau_1 \ldots . \lambda x_n : \tau_n . \kappa$ where κ is a fresh, unknown refinement variable that must be solved to a boolean valued expression over $x_1, \ldots, x_n$. Thus, we generate a *wellformedness* constraint $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \kappa$ and carry out typechecking with template, which, as before, yields implication constraints over κ, which can, as before, be solved via predicate abstraction. Finally, in each refinement template, we replace each κ with its solution e_κ to get the inferred refinement instantiations.

Table 3.1. (LOC) is the number of non-comment Haskell source code lines as reported by *sloccount*, **(Specs)** is the number of lines of type specifications, **(Annot)** is the number of lines of other annotations, including refined datatype definitions, type aliases and measures, required for verification, **(Time)** is the time in seconds taken for verification.

Program	LOC	Specs	Annot	Time (s)
`Micro`	32	19	4	2
`Vector`	56	56	2	14
`ListSort`	29	4	1	3
`Data.List.sort`	71	3	1	8
`Data.Set.Splay`	136	15	11	15
`Data.Map.Base`	1399	119	31	235
Total	1723	216	50	277

3.3 Evaluation

In this section, we empirically evaluate the expressiveness and usability of abstract refinement types by implementing abstract refinement in LIQUID HASKELL as uninterpreted functions. We use LIQUID HASKELL to typecheck a set of challenging benchmark programs. (We defer the task of extending the metatheory to a call-by-name calculus to future work).

Benchmarks We have evaluated LIQUID HASKELL over the following list of benchmarks which, in total, represent the different kinds of reasoning described in § 3.1. While we can prove, and previously have proved [47], many so-called "functional correctness" properties of these data structures using refinement types, in this work we focus on the key invariants which are captured by abstract refinements.

- `Micro`, which includes several functions demonstrating parametric reasoning with base values, type classes, and higher-order loop invariants for traversals and folds, as described in § 3.1.1 and § 3.1.4;

- `Vector`, which includes the domain- and range-generic `Vec` functions and several "clients" that use the generic `Vec` to implement incremental initialization, null-terminated strings, and memoization, as described in § 3.1.2;

- `ListSort`, which includes various textbook sorting algorithms including insert-, merge- and quick-sort. We verify that the functions actually produce sorted lists, *i.e.* are of type

`IncrList a`, as described in § 3.1.3;

- `Data.List.sort`, which includes three non-standard, optimized list sorting algorithms, as found in the `base` package. These employ lists that are increasing and decreasing, as well as lists of (sorted) lists, but we can verify that they also finally produce values of type `IncrList a`;

- `Data.Set.Splay`, which is a purely functional, top-down splay set library from the `llrbtree` package. We verify that all the interface functions take and return binary search trees;

- `Data.Map.Base`, which is the widely-used implementation of functional maps from the `containers` package. We verify that all the interface functions preserve the crucial binary search ordering property and various related invariants.

Table 3.1 quantitatively summarizes the results of our evaluation. We now give a qualitative account of our experience using LIQUID HASKELL by discussing what the specifications and other annotations look like.

Specifications are usually simple In our experience, abstract refinements greatly simplify writing specifications for the *majority* of interface or public functions. For example, for `Data.Map.Base`, we defined the refined version of the `Tree` ADT (actually called `Map` in the source, we reuse the type from § 3.1.3 for brevity), and then instantiated it with the concrete refinements for binary-search ordering with the alias `BST k v` as described in § 3.1.3. Most refined specifications were just the Haskell types with the `Tree` type constructor replaced with the alias `BST`. For example, the type of `fromList` is refined from `(Ord k) ⇒ [(k, a)] → Tree k a` to `(Ord k) ⇒ [(k, a)] → BST k a`. Furthermore, intra-module Liquid type inference permits the automatic synthesis of necessary stronger types for private functions.

Auxiliary Invariants are sometimes Difficult However, there are often rather thorny *internal* functions with tricky invariants, whose specification can take a bit of work. For example, the function `trim` in `Data.Map.Base` has the following behavior (copied verbatim from the

documentation): "`trim blo bhi t` trims away all subtrees that surely contain no values between the range `blo` to `bhi`. The returned tree is either empty or the key of the root is between `blo` and `bhi`." Furthermore `blo` (resp. `bhi`) are specified as option (*i.e.* `Maybe`) values with `Nothing` denoting $-\infty$ (resp. $+\infty$). Fortunately, refinements suffice to encode such properties. First, we define measures

```
measure isJust       :: Maybe a → Bool
  isJust (Just x)    = true
  isJust (Nothing)   = false

measure fromJust     :: Maybe a → a
  fromJustS (Just x) = x

measure isBin         :: Tree k v → Bool
  isBin (Bin _ _ _ _) = true
  isBin (Tip)         = false

measure key :: Tree k v → k
  key (Bin k _ _ _)   = k
```

which respectively embed the `Maybe` and `Tree` root value into the refinement logic, after which we can type the `trim` function as

```
trim :: (Ord k) ⇒ blo:Maybe k
                → bhi:Maybe k
                → BST k a
                → {v:BST k a | bound(blo, v, bhi)}
```

where `bound` is simply a refinement alias

```
refinement bound(lo, v, hi)
  =  isBin(v) ⇒ isJust(lo) ⇒ fromJust(lo) < key(v)
  ∧  isBin(v) ⇒ isJust(hi) ⇒ fromJust(hi) > key(v)
```

That is, the output refinement states that the root is appropriately lower- and upper- bounded if the relevant terms are defined. Thus, refinement types allow one to formalize the crucial behavior as machine-checkable documentation.

Code Modifications On a few occasions we also have to change the code slightly, typically to make explicit values on which various invariants depend. Often, this is for a trivial reason; a simple re-ordering of binders so that refinements for *later* binders can depend on earlier ones. Sometimes we need to introduce "ghost" values so we can write the specifications (*e.g.* the `foldr` in § 3.1.4). Another example is illustrated by the use of list `append` in `quickSort`. Here, the `append` only produces a sorted list if the two input lists are sorted and such that each element in the first is less than each element in the second. We address this with a special `append` parameterized on `pivot`

```
append :: pivot:a
       → IncrList {v:a | v < pivot}
       → IncrList {v:a | v > pivot}
       → IncrList a
append pivot [] ys     = pivot : ys
append pivot (x:xs) ys = x : append pivot xs ys
```

3.4 Conclusion

We presented *abstract refinement types* which enable quantification over the refinements of data- and function-types. Our key insight is that we can avail of quantification while preserving SMT-based decidability, simply by encoding refinement parameters as *uninterpreted* propositions within the refinement logic. We showed how this mechanism yields a variety of sophisticated means for reasoning about programs, including: *parametric* refinements for reasoning with type classes, *index-dependent* refinements for reasoning about key-value maps, *recursive* refinements for reasoning about recursive data types, and *inductive* refinements for reasoning about higher-order traversal routines. We implemented our approach in LIQUID HASKELL and present experiments using our tool to verify correctness invariants of various programs.

As discussed in 3.3, verification many times required code modifications and definition of "ghost" variables (*e.g.* to verify `append`), that is, extra arguments not used at run time but required for specifications. In next chapter we raise this limitation by introducing Bounded Refinement Types, that impose constrains in the abstract refinements to further increase the

expressiveness of decidable specifications. Abstract and Bounded Refinement Types lead to a relatively complete [94] specification system, that is the system can express any specification (relatively to the underlying logic) without the requirement of code modifications and "ghost" variables.

Acknowledgments The material of this chapter are adapted from the following publication: N. Vazou, P. Rondon, and R. Jhala, "Abstract Refinement Types", ESOP, 2013.

Chapter 4

Bounded Refinement Types

> Problems are hidden opportunities, and constraints can actually boost creativity.
>
> *– Martin Villeneuve*

In this chapter we introduce *Bounded Refinement Types* which enable *bounded quantification* over refinements. Previously (Chapter 3), we developed Abstract Refinement Types, a mechanism for quantifying type signatures over abstract refinement parameters. We preserved decidability of checking and inference by encoding abstractly refined types with uninterpreted functions obeying the decidable axioms of congruence [69]. While useful, refinement quantification was not enough to enable higher order abstractions requiring fine grained *dependencies between* abstract refinements. In this chapter, we solve this problem by enriching signatures with bounded quantification. The *bounds* correspond to Horn implications between abstract refinements, which, as in the classical setting, correspond to subtyping constraints that must be satisfied by the concrete refinements used at any call-site. This addition proves to be remarkably effective.

- First, we demonstrate via a series of short examples how bounded refinements enable the specification and verification of diverse textbook higher order abstractions that were hitherto beyond the scope of decidable refinement typing (§ 4.1).

- Second, we formalize bounded types and show how bounds are translated into "ghost" functions, reducing type checking and inference to the "unbounded" setting of chapter 3,

thereby ensuring that checking remains decidable. Furthermore, as the bounds are Horn constraints, we can directly reuse the abstract interpretation of Liquid Typing [79] to automatically infer concrete refinements at instantiation sites (§ 4.2).

- Third, to demonstrate the expressiveness of bounded refinements, we use them to build a typed library for extensible dictionaries, to then implement a relational algebra library on top of those dictionaries, and to finally build a library for type-safe database access (§ 4.3).

- Finally, we use bounded refinements to develop a *Refined State Transformer* monad for stateful functional programming, based upon Filliâtre's method for indexing the monad with pre- and post-conditions [32]. We use bounds to develop branching and looping combinators whose types signatures capture the derivation rules of Floyd-Hoare logic, thereby obtaining a library for writing verified stateful computations (§ 4.4). We use this library to develop a refined IO monad that tracks capabilities at a fine-granularity, ensuring that functions only access specified resources (§ 4.5).

We have implemented Bounded Refinement Types in LIQUID HASKELL. The source code of the examples (with slightly more verbose concrete syntax) is at [90]. While the construction of these verified abstractions is possible with full dependent types, bounded refinements keep checking automatic and decidable, use abstract interpretation to automatically synthesize refinements (*i.e.* pre- and post-conditions and loop invariants), and most importantly enable retroactive or *gradual* verification as when erase the refinements, we get valid programs in the host language. Thus, bounded refinements point a way towards both automated and expressive verification.

4.1 Overview

We start with a high level overview of bounded refinement types. We first present a short introduction to refinement type specifications, to make this chapter self contained. Then, we introduce bounded refinements, and show how they permit *modular* higher-order specifications. Finally, we describe how they are implemented via an elaboration process that permits *automatic*

first-order verification.

4.1.1 Preliminaries

Refinement Types let us precisely specify subsets of values, by conjoining base types with logical predicates that constrain the values. We get decidability of type checking, by limiting these predicates to decidable, quantifier-free, first-order logics, including the theory of linear arithmetic, uninterpreted functions, arrays, bit-vectors and so on. For example, the refinement types

```
type Pos     = {v:Int | 0 < v}
type IntGE x = {v:Int | x ≤ v}
```

specify subsets of `Int` corresponding to values that are positive or larger than some other value `x` respectively. We can use refinement types to specify contracts like pre- and post-conditions by suitably refining the input and output types of functions.

Preconditions are specified by refining input types. We specify that the function `assert` must *only* be called with `True`, where the type `TRUE` contains only the singleton `True`:

```
type TRUE = {v:Bool | v  ⇔  True}


assert          :: TRUE → a → a
assert True x   = x
assert False _ = error "Provably Dead Code"
```

We can specify post-conditions by refining output types. For example, a primitive `Int` comparison operator `leq` can be assigned a type that says that the output is `True` iff the first input is actually less than or equal to the second:

```
leq :: x:Int → y:Int → {v:Bool | v  ⇔ x ≤ y}
```

Refinement Type Checking proceeds by checking that at each application, the types of the actual arguments are *subtypes* of those of the function inputs, in the environment (or context) in which the call occurs. Consider the function:

```
checkGE     :: a:Int → b:IntGE a → Int
checkGE a b = assert cmp b
  where cmp = a `leq` b
```

To verify the call to `assert` we check that the actual parameter `cmp` is a subtype of `TRUE`, under the assumptions given by the input types for `a` and `b`. Via subtyping [100] the check reduces to establishing the validity of the *verification condition* (VC)

```
a ≤ b ⇒ (cmp ⇔ a ≤ b) ⇒ v = cmp ⇒ (v ⇔ true)
```

The first antecedent comes from the input type of `b`, the second from the type of `cmp` obtained from the output of `leq`, the third from the *actual* input passed to `assert`, and the goal comes from the input type *required* by `assert`. An SMT solver [69] readily establishes the validity of the above VC, thereby verifying `checkGE`.

4.1.2 Bounded Refinements

Refinement types hit various expressiveness walls, as for decidability, refinements are constraint to first order, decidable logics. Consider the following example from [94]. `find` takes as input a predicate `q`, a continuation `k` and a starting number `i`; it proceeds to compute the smallest `Int` (larger than `i`) that satisfies `q`, and calls `k` with that value. `ex1` passes `find` a continuation that checks that the "found" value equals or exceeds `n`.

```
ex1 :: (Int → Bool) → Int → ()
ex1 q n = find q (checkGE n) n


find q k i
  | q i       = k i
  | otherwise = find q k (i + 1)
```

Verification fails as there is no way to specify that `k` is only called with arguments greater than `n`. First, the variable `n` is not in scope at the function definition so we cannot refer to it. Second, we could try to say that `k` is invoked with values greater than or equal to `i`, which gets substituted with `n` at the call-site. Alas, due to the currying order, `i` too is not in scope at the point where `k`'s type is defined so the type for `k` cannot depend upon `i`.

Can Abstract Refinements Help? Lets try to use Abstract Refinements, from chapter 3, to abstract over the refinement that `i` enjoys, and assign `find` the type:

```
find :: (Int → Bool) → (Int<p> → a) → Int<p> → a
```

which states that for any refinement `p`, the function takes an input `i` which satisfies `p` and hence that the continuation is also only invoked on a value which trivially enjoys `p`, namely `i`. At the call-site in `ex1` we can instantiate

$$\texttt{p} \mapsto \lambda \texttt{v} \rightarrow \texttt{n} \leq \texttt{v} \tag{4.1}$$

This instantiated refinement is satisfied by the parameter `n` and is sufficient to verify, via function subtyping, that `checkGE n` will only be called with values satisfying `p`, and hence larger than `n`.

The function find is ill-typed as the signature requires that at the recursive call site, the value `i+1` *also* satisfies the abstract refinement `p`. While this holds for the example we have in mind (4.1), it does not hold *for all* `p`, as required by the type of `find`! Concretely, `{v:Int | v = i + 1}` is in general *not* a subtype of `Int<p>`, as the associated VC

$$\ldots \Rightarrow \texttt{p i} \Rightarrow \texttt{p (i+1)} \tag{4.2}$$

is *invalid* – the type checker thus (soundly!) rejects `find`.

We must Bound the Quantification of `p` to limit it to refinements satisfying some constraint, in this case that `p` is *upward closed*. In the dependent setting, where refinements may refer to program values, bounds are naturally expressed as constraints between refinements. We define a bound, `UpClosed` which states that `p` is a refinement that is *upward closed*, *i.e.* satisfies $\forall$ `x. p x ⇒ p (x+1)`, and use it to type `find` as:

```
bound UpClosed (p :: Int → Bool)
  = \x → p x ⇒ p (x+1)

find :: (UpClosed p) ⇒ (Int → Bool)
                     → (Int<p> → a)
                     →  Int<p> → a
```

This time, the checker is able to use the bound to verify the VC (4.2). We do so in a way that refinements (and thus VCs) remain quantifier free and hence, SMT decidable (§ 4.1.4).

At the call to `find` in the body of `ex1`, we perform the instantiation (4.1) which generates the

additional VC `n ≤ x ⇒ n ≤ x+1` by plugging in the concrete refinements to the bound constraint. The SMT checks the validity of the VC and hence this instantiation, thereby statically verifying `ex1`, *i.e.* that the assertion inside `checkGE` cannot fail.

4.1.3 Bounds for Higher-Order Functions

Next, we show how bounds expand the scope of refinement typing by letting us write precise modular specifications for various canonical higher-order functions.

Function Composition

First, consider `compose`. What is a modular specification for `compose` that would let us verify that `ex2` enjoys the given specification?

```
compose f g x = f (g x)

type Plus x y = {v:Int | v = x + y}

ex2    :: n:Int → Plus n 2
ex2    = incr `compose` incr

incr   :: n:Int → Plus n 1
incr n = n + 1
```

The challenge is to chain the dependencies between the input and output of `g` and the input and output of `f` to obtain a relationship between the input and output of the composition. We can capture the notion of chaining in a bound:

```
bound Chain p q r = \x y z →
      q x y ⇒ p y z ⇒ r x z
```

which states that for any `x`, `y` and `z`, if (1) `x` and `y` are related by `q`, and (2) `y` and `z` are related by `p`, then (3) `x` and `z` are related by `r`.

We use `Chain` to type `compose` using three abstract refinements `p`, `q` and `r`, relating the arguments and return values of `f` and `g` to their composed value. (Here, `c<r x>` abbreviates `{v:c | r x v}`).

```
compose :: (Chain p q r) ⇒ (y:b → c<p y>)
                         → (x:a → b<q x>)
                         → (w:a → c<r w>)
```

To verify `ex2` we instantiate, at the call to `compose`,

```
p, q ⊢> \x v → v = x + 1
   r ⊢> \x v → v = x + 2
```

The above instantiation satisfies the bound, as shown by the validity of the VC derived from instantiating `p`, `q`, and `r` in `Chain`:

```
y = x + 1 ⇒ z = y + 1 ⇒ z == x + 2
```

and hence, we can check that `ex2` implements its specified type.

List Filtering

Next, consider the list `filter` function. What type signature for `filter` would let us check `positives`?

```
filter q (x:xs)
  | q x          = x : filter q xs
  | otherwise    = filter q xs
filter _ []      = []

positives        :: [Int] → [Pos]
positives        = filter isPos
  where isPos x = 0 < x
```

Such a signature would have to relate the `Bool` returned by `f` with the property of the `x` that it checks for. Typed Racket's latent predicates [91] account for this idiom, but are a special construct limited to `Bool`-valued "type" tests, and not arbitrary invariants. Another approach is to avoid the so-called "Boolean Blindness" that accompanies `filter` by instead using option types and `mapMaybe`.

We overcome blindness using a witness bound:

```
bound Witness p w = \x b → b ⇒ w x b ⇒ p x
```

which says that `w` *witnesses* the refinement `p`. That is, for any boolean `b` such that `w x b` holds, if `b` is `True` then `p x` also holds.

We can give `filter` a type that says that the output values enjoy a refinement `p` as long as the test predicate `q` returns a boolean witnessing `p`:

```
filter :: (Witness p w) ⇒ (x:a → Bool<w x>)
                        → List a
                        → List a<p>
```

To verify `positives` we infer the following type and instantiations for the abstract refinements `p` and `w` at the call to `filter`:

```
isPos :: x:Int → {v:Bool | v ⇔ 0 < x}
p     ⊢> \v    → 0 < v
w     ⊢> \x b  → b ⇔ 0 < x
```

List Folding

Next, consider the list fold-right function. Suppose we wish to prove the following type for ex3:

```
foldr :: (a → b → b) → b → List a → b
foldr op b []     = b
foldr op b (x:xs) = x `op` foldr op b xs

ex3 :: xs:List a → {v:Int | v == len xs}
ex3 = foldr (\_ → incr) 0
```

where `len` is a *logical* or *measure* function used to represent the number of elements of the list in the refinement logic 2.1.6:

```
measure len   :: List a → Nat
  len []      = 0
  len (x:xs)  = 1 + len xs
```

We specify induction as a bound. Let (1) `inv` be an abstract refinement relating a list `xs` and the result `b` obtained by folding over it and (2) `step` be an abstract refinement relating the inputs `x`, `b`

and output `b′` passed to and obtained from the accumulator `op` respectively. We state that `inv` is closed under `step` as:

```
bound Inductive inv step = \x xs b b′ →
      inv xs b ⇒ step x b b′ ⇒ inv (x:xs) b′
```

We can give `foldr` a type that says that the function *outputs* a value that is built inductively over the entire *input* list:

```
foldr :: (Inductive inv step)
      ⇒ (x:a → acc:b → b<step x acc>)
      → b<inv []>
      → xs:List a
      → b<inv xs>
```

That is, for any invariant `inv` that is inductive under `step`, if the initial value `b` is `inv`-related to the empty list, then the folded output is `inv`-related to the input list `xs`.

We verify `ex3` by inferring, at the call to `foldr`

```
inv  ⊢> \xs v   → v  == len xs
step ⊢> \x b b′ → b′ == b + 1
```

The SMT solver validates the VC obtained by plugging the above into the bound. Instantiating the signature for `foldr` yields precisely the output type desired for `ex3`.

Previously, 3 describes a way to type `foldr` using abstract refinements that required the operator `op` to have one extra ghost argument. Bounds let us express induction without ghost arguments.

4.1.4 Implementation

To implement bounded refinement typing, we must solve two problems. Namely, how do we (1) *check* and (2) *use* functions with bounded signatures? We solve both problems via an insight inspired by the way typeclasses are implemented in Haskell.

1. ***A Bound Specifies*** a function type whose inputs are unconstrained and whose output is some value that carries the refinement corresponding to the bound's body.

2. ***A Bound is Implemented*** by a ghost function that returns `True`, but is defined in a context where the bound's constraint holds when instantiated to the concrete refinements at the context.

We elaborate bounds into ghost functions satisfying the bound's type. To *check* bounded functions, we need to *call* the ghost function to materialize the bound constraint at particular values of interest. Dually, to *use* bounded functions, we need to *create* ghost functions whose outputs are guaranteed to satisfy the bound constraint. This elaboration reduces *bounded* refinement typing to the simpler problem of *unbounded* abstract refinement typing. The formalization of our elaboration is described in § 4.2. Next, we illustrate the elaboration by explaining how it addresses the problems of checking and using bounded signatures like `compose`.

We Translate Bounds into Function Types called the bound-type where the inputs are unconstrained, and the outputs satisfy the bound's constraint. For example, the bound `Chain` used to type `compose` in § 4.1.3, corresponds to a function type, yielding the translated type for `compose`:

```
type ChainTy p q r
  =  x:a → y:b → z:c → {v:Bool | q x y ⇒ p y z ⇒ r x z}

compose :: (ChainTy p q r)
        → (y:b → c<p y>)
        → (x:a → b<q x>)
        → (w:a → c<r w>)
```

To Check Bounded Functions we view the bound constraints as extra (ghost) function parameters (cf. type class dictionaries), that satisfy the bound-type. Crucially, each expression where a subtyping constraint would be generated (by plain refinement typing) is wrapped with a "call" to the ghost to materialize the constraint at values of interest. For example we elaborate `compose` into:

```
compose $chain f g x =
  let t1 = g x
      t2 = f t1
      _  = $chain x t1 t2   -- materialize
```

```
in  t2
```

In the elaborated version `$chain` is the ghost parameter corresponding to the bound. As is standard [79], we perform ANF-conversion to name intermediate values, and then wrap the function output with a call to the ghost to materialize the bound's constraint. Consequently, the output of compose, namely `t2`, is checked to be a subtype of the specified output type, in an environment *strengthened* with the bound's constraint instantiated at `x`, `t1` and `t2`. This subtyping reduces to a quantifier free VC:

```
    q x t1
⇒   p t1 t2
⇒ (q x t1 ⇒ p t1 t2 ⇒ r x t2)
⇒   v = t2 ⇒ r x v
```

whose first two antecedents are due to the types of `t1` and `t2` (via the output types of `g` and `f` respectively) and the third comes from the call to `$chain`. The output value `v` has the singleton refinement that states it equals to `t2` and finally the VC states that the output value `v` must be related to the input `x` via `r`. An SMT solver validates this decidable VC easily, thereby verifying `compose`.

Our elaboration inserts materialization calls *for all* variables (of the appropriate type) that are in scope at the given point. This could introduce upto n^k calls where k is the number of parameters in the bound and n the number of variables in scope. In practice (*e.g.* in `compose`) this number is small (*e.g.* 1) since we limit ourselves to variables of the appropriate types.

To preserve semantics we ensure that none of these materialization calls can diverge, by carefully constraining the structure of the arguments that instantiate the ghost functional parameters.

At Uses of Bounded Functions our elaboration uses the bound-type to create lambdas with appropriate parameters that just return `true`. For example, `ex2` is elaborated to:

```
ex2 = compose (\_ _ _ → true) incr incr
```

This elaboration seems too naïve to be true: how do we ensure that the function actually satisfies the bound type?

Happily, that is automatically taken care of by function subtyping. Recalling the translated type for `compose`, the elaborated lambda `(\_ _ _ → true)` is constrained to be a subtype of `ChainTy p q r`. In particular, given the call site instantiation

```
p ↦ \ y z → z == y + 1
q ↦ \ x y → y == x + 1
r ↦ \ x z → z == x + 2
```

this subtyping constraint reduces to the quantifier-free VC:

$$[\![\Gamma]\!] \Rightarrow \texttt{true} \Rightarrow \texttt{(z == y + 1)} \Rightarrow \texttt{(y == x + 1)} \Rightarrow \texttt{(z == x + 2)}$$

where Γ contains assumptions about the various binders in scope. The above VC is easily proved valid by an SMT solver, thereby verifying the subtyping obligation defined by the bound, and hence, that `ex2` satisfies the given type.

4.2 Formalism

Next we formalize Bounded Refinement Types by defining a core calculus λ_B and showing how it can be reduced to λ_P, the core language of Abstract Refinement Types 3. We start by defining the syntax and semantics of λ_P (§ 4.2.1) and the syntax of λ_B (§ 4.2.2). Next, we provide a translation from λ_B to λ_P (§ 4.2.3). Then, we prove soundness by showing that our translation is semantics preserving (§ 4.2.4). Finally, we describe how type inference remains decidable in the presence of bounded refinements (§ 4.2.5).

4.2.1 Syntax of λ_P

We build our core language on top of λ_P, the language of Abstract Refinement Types3. Figure 4.1 summarizes the syntax of λ_P, a polymorphic λ-calculus extended with abstract refinements. For an easier transition to the syntax of Bounded Refinement Types, we rewrite the syntax of λ_P as initially presented in Figure 3.1 by stratifying type schemata to include bounded types, that for now, are plain types.

The Expressions of λ_P include the usual variables x, primitive constants c, λ-abstraction $\lambda x : t.e$,

$$
\begin{array}{rrcl}
\textbf{\textit{Expressions}} & e & ::= & x \mid c \mid \lambda x:t.e \mid e\ x \mid \mathsf{let}\ x{:}t = e\ \mathsf{in}\ e \\
 & & \mid & \Lambda\alpha.e \mid e[t] \mid \Lambda\pi:t.e \mid e[\phi] \\
\textbf{\textit{Constants}} & c & ::= & \texttt{True} \mid \texttt{False} \mid \texttt{Crash} \mid 0 \mid 1 \mid -1 \mid \ldots \\
\textbf{\textit{Parametric Refinements}} & \phi & ::= & r \mid \lambda x:b.\phi \\
\textbf{\textit{Predicates}} & p & ::= & c \mid \neg p \mid p = p \mid \ldots \\
\textbf{\textit{Atomic Refinements}} & a & ::= & p \mid \pi\,\bar{x} \\
\textbf{\textit{Refinements}} & r & ::= & a \mid a \wedge r \mid a \Rightarrow r \\
\textbf{\textit{Basic Types}} & b & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{a} \\
\textbf{\textit{Types}} & t & ::= & \{v:b \mid r\} \mid \{v:(x:t) \to t \mid r\} \\
\textbf{\textit{Bounded Types}} & \rho & ::= & t \\
\textbf{\textit{Schemata}} & \sigma & ::= & \rho \mid \forall\alpha.\sigma \mid \forall\pi:t.\sigma
\end{array}
$$

Figure 4.1. Stratified Syntax of λ_P.

application $e\ e$, let bindings $\mathsf{let}\ x{:}t = e\ \mathsf{in}\ e$, type abstraction $\Lambda\alpha.e$, and type application $e[t]$. (We add let-binders to λ_P from Figure 3.1 as they can be reduced to λ-abstractions in the usual way). The parameter t in the type application is a *refinement type*, as described shortly. Finally, λ_P includes refinement abstraction $\Lambda\pi:t.e$, which introduces a refinement variable π (with its type t), which can appear in refinements inside e, and the corresponding refinement application $e[\phi]$ that substitutes an abstract refinement with the parametric refinement ϕ, *i.e.* refinements r closed under lambda abstractions.

The Primitive Constants of λ_P include `True`, `False`, `0`, `1`, `-1`, *etc.*. In addition, we include a special untypable constant `Crash` that models "going wrong". Primitive operations return a crash when invoked with inputs outside their domain, *e.g.* when `/` is invoked with `0` as the divisor or when an `assert` is applied to `False`.

Atomic Refinements a are either concrete or abstract refinements. A *concrete refinement* p is a boolean valued expression (such as a constant, negation, equality, *etc.*) drawn from a *strict subset* of the language of expressions which includes only terms that (a) neither diverge nor crash and (b) can be embedded into an SMT decidable logic including the quantifier free theory of linear arithmetic and uninterpreted functions [100]. An *abstract refinement* $\pi\,\bar{x}$ is an application of a refinement variable π to a sequence of program variables. A *refinement* r is either a conjunction or implication of atomic refinements. To enable inference, we only allow implications to appear

$$\begin{array}{rrcl} \textbf{\textit{Bounded Types}} & \rho & ::= & t \mid \{\phi\} \Rightarrow \rho \\ \textbf{\textit{Expressions}} & e & ::= & \ldots \mid \Lambda\{\phi\}.e \mid e\{\phi\} \end{array}$$

Figure 4.2. Extending Syntax of λ_P to λ_B.

within bounds ϕ (§ 4.2.5).

The Types of λ_P, written t, include basic types, dependent functions and schemata quantified over type and refinement variables α and π respectively. A basic type is one of `Int`, `Bool`, or a type variable α. A refined type t is either a refined basic type $\{v : b \mid r\}$, or a dependent function type $\{v : (x : t) \to t \mid r\}$ where the parameter x can appear in the refinements of the output type. (We include refinements for functions, as refined type variables can be replaced by function types. However, typechecking ensures these refinements are trivially true). In λ_P bounded types ρ are just a synonym for types t. Finally, schemata quantify bounded types over type and refinement variables.

4.2.2 Syntax of λ_B

Figure 4.2 shows how we obtain the syntax for λ_B by extending the syntax of λ_P with *bounded* types.

The Types of λ_B extend those of λ_P with bounded types ρ, which are the types t guarded by bounds ϕ.

The Expressions of λ_B extend those of λ_P with *abstraction* over bounds $\Lambda\{\phi\}.e$ and *application* of bounds $e\{\phi\}$. Intuitively, if an expression e has some type ρ then $\Lambda\{\phi\}.e$ has the type $\{\phi\} \Rightarrow \rho$. We include an explicit bound application form $e\{\phi\}$ to simplify the formalization; these applied bounds are automatically synthesized from the type of e, and are of the form $\overline{\lambda x : \rho.\texttt{True}}$.

Notation. We write b, $b\langle\pi\ \bar{x}\rangle$, and $\{v : b\langle\pi\ \bar{x}\rangle \mid r\}$ to abbreviate $\{v : b \mid \texttt{True}\}$, $\{v : b \mid \pi\ \bar{x}\ v\}$, and $\{v : b \mid r \wedge \pi\ \bar{x}\ v\}$ respectively. We say a type or schema is *non-refined* if all the refinements in it

are *True*. We get the *shape* of a type t (*i.e.* the System-F type) by the function $\mathsf{Shape}(t)$ defined:

$$\mathsf{Shape}(\{v : b \mid r\}) \doteq b$$

$$\mathsf{Shape}(\{v : (x : t_1) \to t_2 \mid r\}) \doteq \mathsf{Shape}(t_1) \to \mathsf{Shape}(t_2)$$

4.2.3 Translation from λ_B to λ_P

Next, we show how to translate a term from λ_B to one in λ_P. We assume, without loss of generality that the terms in λ_B are in Administrative Normal Form (*i.e.* all applications are to variables).

Bounds Correspond To Functions that explicitly "witness" the fact that the bound constraint holds at a given set of "input" values. That is we can think of each bound as a universally quantified relationship between various (abstract) refinements; by "calling" the function on a set of input values $x_1, \ldots, x_n$, we get to *instantiate* the constraint for that particular set of values.

Bound Environments Φ are used by our translation to track the set of bound-functions (names) that are in scope at each program point. These names are distinct from the regular program variables that will be stored in Variable Environments Γ. We give bound functions distinct names so that they cannot appear in the regular source, only in the places where calls are inserted by our translation. The translation ignores refinements entirely; both environments map their names to their non-refined types.

The Translation is formalized in Figure 4.3 via a relation $\Gamma; \Phi \vdash e \rightsquigarrow e'$ that translates the expression e in λ_B into e' in λ_P. Most of the rules in figure 4.3 recursively translate the sub-expressions. Types that appear inside expressions are syntactically restricted to not contain bounds, thus types inside expressions do not require translation. Here we focus on the three interesting rules:

1. ***At bound abstractions*** $\Lambda\{\phi\}.e$ we convert the bound ϕ into a bound-function parameter of a suitable type,

2. ***At variable binding sites*** *i.e.* λ- or let-bindings, we *use* the bound functions to *materialize*

$$\textit{Variable Environment} \quad \Gamma ::= \emptyset \mid \Gamma, x:\tau$$
$$\textit{Bound Environment} \quad \Phi ::= \emptyset \mid \Phi, x:\tau$$

Translation $\boxed{\Gamma;\Phi \vdash e \rightsquigarrow e}$

$$\frac{}{\Gamma;\Phi \vdash x \rightsquigarrow x}\ \textsc{Var} \qquad \frac{}{\Gamma;\Phi \vdash c \rightsquigarrow c}\ \textsc{Con}$$

$$\frac{\Gamma' = \Gamma, x:\mathsf{Shape}(t) \qquad \Gamma';\Phi \vdash e \rightsquigarrow e'}{\Gamma;\Phi \vdash \lambda x:t.e \rightsquigarrow \lambda x:t.\mathsf{Inst}(\Gamma',\Phi,e')}\ \textsc{Fun}$$

$$\frac{\Gamma;\Phi \vdash e_x \rightsquigarrow e'_x \qquad \Gamma' = \Gamma, x:\mathsf{Shape}(t) \qquad \Gamma';\Phi \vdash e \rightsquigarrow e'}{\Gamma;\Phi \vdash \mathsf{let}\ x:t = e_x\ \mathsf{in}\ e \rightsquigarrow \mathsf{let}\ x:\tau = e'_x\ \mathsf{in}\ \mathsf{Inst}(\Gamma',\Phi,e')}\ \textsc{Let}$$

$$\frac{\Gamma;\Phi \vdash e_1 \rightsquigarrow e'_1 \qquad \Gamma;\Phi \vdash e_2 \rightsquigarrow e'_2}{\Gamma;\Phi \vdash e_1\ e_2 \rightsquigarrow e'_1\ e'_2}\ \textsc{App}$$

$$\frac{\Gamma;\Phi \vdash e \rightsquigarrow e'}{\Gamma;\Phi \vdash \Lambda\alpha.e \rightsquigarrow \Lambda\alpha.e'}\ \textsc{TAbs} \qquad \frac{\Gamma;\Phi \vdash e \rightsquigarrow e'}{\Gamma;\Phi \vdash e\,[t] \rightsquigarrow e'\,[t]}\ \textsc{TApp}$$

$$\frac{\Gamma;\Phi \vdash e \rightsquigarrow e'}{\Gamma;\Phi \vdash \Lambda\pi:t.e \rightsquigarrow \Lambda\pi:t.e'}\ \textsc{PAbs} \qquad \frac{\Gamma;\Phi \vdash e_1 \rightsquigarrow e'_2 \qquad \Gamma;\Phi \vdash e_1 \rightsquigarrow e'_2}{\Gamma;\Phi \vdash e_1\,[e_2] \rightsquigarrow e'_1\,[e'_2]}\ \textsc{PApp}$$

$$\frac{\text{fresh } f \qquad \Gamma;\Phi, f{:}\mathsf{Shape}(\langle\!|\phi|\!\rangle) \vdash e \rightsquigarrow e'}{\Gamma;\Phi \vdash \Lambda\{\phi\}.e \rightsquigarrow \lambda f:\langle\!|\phi|\!\rangle.e'}\ \textsc{CAbs} \qquad \frac{\Gamma;\Phi \vdash e \rightsquigarrow e'}{\Gamma;\Phi \vdash e\{\phi\} \rightsquigarrow e'\ \mathsf{Const}(\phi)}\ \textsc{CApp}$$

Figure 4.3. Translation Rules from λ_B to λ_P.

the bound constraints for all the variables in scope after the binding,

3. ***At bound applications*** $e\{\phi\}$ we *provide* regular functions that witness that the bound constraints hold.

1. Rule **CABS** translates bound abstractions $\Lambda\{\phi\}.e$ into a plain λ-abstraction. In the translated expression $\lambda f:\langle\!|\phi|\!\rangle.e'$ the bound becomes a function named f with type $\langle\!|\phi|\!\rangle$ defined:

$$\langle\!|\lambda x:b.\phi|\!\rangle \doteq (x:b) \to \langle\!|\phi|\!\rangle$$
$$\langle\!|r|\!\rangle \doteq \{v:\texttt{Bool} \mid r\}$$

That is, $\langle\!|\phi|\!\rangle$ is a function type whose final output carries the refinement corresponding to the constraint in ϕ. Note that the translation generates a fresh name f for the bound function (ensuring

that it cannot be used in the regular code) and saves it in the bound environment Φ to let us materialize the bound constraint when translating the body e of the abstraction.

***2. Rules* FUN *and* LET** materialize bound constraints at variable binding sites (λ-abstractions and let-bindings respectively). If we view the bounds as universally quantified constraints over the (abstract) refinements, then our translation exhaustively and eagerly *instantiates* the constraints at each point that a new binder is introduced into the variable environment, over all the possible candidate sets of variables in scope at that point. The instantiation is performed by $\mathsf{Inst}(\Gamma, \Phi, e)$

$$
\begin{array}{rcl}
\mathsf{Inst}(\Gamma, \Phi, e) & \doteq & \mathsf{Wrap}(e, \mathsf{Instances}(\Gamma, \Phi)) \\
\mathsf{Wrap}(e, \{e_1, \ldots, e_n\}) & \doteq & \mathsf{let}\ t_1 = e_1\ \mathsf{in}\ \ldots \mathsf{let}\ t_n = e_n\ \mathsf{in}\ e \\
 & & \text{where } t_i \text{ are fresh } \texttt{Bool} \text{ binders} \\
\mathsf{Instances}(\Gamma, \Phi) & \doteq & \{\ f\ \overline{x} \mid f : \tau \leftarrow \Phi,\ \overline{x : \_} \leftarrow \Gamma,\ \Gamma, f : \tau \vdash_B f\ \overline{x} : \texttt{Bool}\ \}
\end{array}
$$

The function takes the environments Γ and Φ, an expression e and a variable x of type t and uses let-bindings to materialize all the bound functions in Φ that accept the variable x. Here, $\Gamma \vdash_B e : \tau$ is the standard typing derivation judgment for the non-refined System F and so we elide it for brevity.

***3. Rule* CAPP** translates bound applications $e\{\phi\}$ into plain λ abstractions that witness that the bound constraints hold. That is, as within e, bounds are translated to a bound function (parameter) of type $\langle\!|\phi|\!\rangle$, we translate ϕ into a λ-term that, via subtyping must have the required type $\langle\!|\phi|\!\rangle$. We construct such a function via $\mathsf{Const}(\phi)$ that depends only on the *shape* of the bound, *i.e.* the non-refined types of its parameters (and not the actual constraint itself).

$$
\begin{array}{rcl}
\mathsf{Const}(r) & \doteq & \mathit{True} \\
\mathsf{Const}(\lambda x : b.\phi) & \doteq & \lambda x : b.\mathsf{Const}(\phi)
\end{array}
$$

This seems odd: it is simply a constant function, how can it possibly serve as a bound? The answer is that subtyping in the translated λ_P term will verify that in the context in which the above constant function is created, the singleton *True* will indeed carry the refinement corresponding to

the bound constraint, making this synthesized constant function a valid realization of the bound function. Recall that in the example `ex2` of the overview (§ 4.1.4) the subtyping constraint that decides if the constant *True* is a valid bound reduces to the equation 4.3 that is a tautology.

4.2.4 Soundness

The Small-Step Operational Semantics of λ_B are defined by extending a similar semantics for λ_P which is a standard call-by-value calculus where abstract refinements are boolean valued functions. Let $\hookrightarrow_P$ denote the transition relation defining the operational semantics of λ_P and $\hookrightarrow_P^\star$ denote the reflexive transitive closure of $\hookrightarrow_P$. We obtain the transition relation $\hookrightarrow_B$:

$$(\Lambda\{\phi\}.e)\{\phi\} \hookrightarrow_B e \qquad\qquad e \hookrightarrow_B e', \text{if } e \hookrightarrow_P e'$$

Let $\hookrightarrow_B^\star$ denote the reflexive transitive closure of $\hookrightarrow_B$.

The Translation is Semantics Preserving in the sense that if a source term e of λ_B reduces to a constant then the translated variant of e' also reduces to the same constant (as show in [95]):

Lemma 1 (Semantics Preservation)**.** *If* $\emptyset;\emptyset \vdash e \rightsquigarrow e'$ *and* $e \hookrightarrow_B^\star c$ *then* $e' \hookrightarrow_P^\star c$.

The Soundness of λ_B follows by combining the above Semantics Preservation Lemma with the soundness of λ_P.

To Typecheck a λ_B ***program*** e we translate it into a λ_P program e' and then type check e' using the rules of Figure 3.2; if the latter check is safe, then we are guaranteed that the source term e will not crash:

Theorem 1 (Soundness)**.** *If* $\emptyset;\emptyset \vdash e \rightsquigarrow e'$ *and* $\emptyset \vdash e' : \sigma$ *then* $e \not\hookrightarrow_B^\star$ `Crash`.

4.2.5 Inference

A critical feature of bounded refinements is that we can automatically synthesize instantiations of the abstract refinements by: (1) generating templates corresponding to the unknown types where fresh variables κ denote the unknown refinements that an abstract refinement parameter

π is instantiated with, (2) generating subtyping constraints over the resulting templates, and (3) solving the constraints via abstract interpretation.

Inference Requires Monotonic Constraints. Abstract interpretation only works if the constraints are *monotonic* [21], which in this case means that the κ variables, and correspondingly, the abstract refinements π must only appear in *positive* positions within refinements (*i.e.* not under logical negations). The syntax of refinements shown in Figure 4.1 violates this requirement via refinements of the form $\pi\,\overline{x} \Rightarrow r$.

We restrict implications to bounds *i.e.* prohibit them from appearing elsewhere in type specifications. Consequently, the implications only appear in the *output* type of the (first order) "ghost" functions that bounds are translated to. The resulting subtyping constraints only have *implications inside super-types*, *i.e.* as:

$$\Gamma \vdash \{v{:}b \mid a\} \preceq \{v{:}b \mid a_1 \Rightarrow \cdots \Rightarrow a_n \Rightarrow a_q\}$$

By taking into account the semantics of subtyping, we can push the antecedents into the environment, *i.e.* transform the above into an equivalent constraint in the form:

$$\Gamma, x_1 :, \{x_1{:}b_1 \mid a_1'\} :, \ldots, x_n :, \{x_n{:}b_n \mid a_n'\} \vdash \{v{:}b \mid a'\} \preceq \{v{:}b \mid a_q'\}$$

where all the abstract refinements variables π (and hence instance variables κ) appear positively, ensuring that the constraints are monotonic, hence permitting inference via Liquid Typing [79].

4.3 A Refined Relational Database

Next, we use bounded refinements to develop a library for relational algebra, which we use to enable generic, type safe database queries. A relational database stores data in *tables*, that are a collection of *rows*, which in turn are *records* that represent a unit of data stored in the table. The tables's *schema* describes the types of the values in each row of the table. For example, the table in Figure 4.1 organizes information about movies, and has the schema:

```
Title:String, Dir:String, Year:Int, Star:Double
```

Table 4.1. Example entries for Movies Database.

Title	Director	Year	Star
"Birdman"	"Iñárritu"	2014	8.1
"Persepolis"	"Paronnaud"	2007	8.0

First, we show how to write type safe extensible records that represent rows, and use them to implement database tables (§ 4.3.1). Next, we show how bounds let us specify type safe relational operations and how they may be used to write safe database queries (§ 4.3.2).

4.3.1 Rows and Tables

We represent the rows of a database with dictionaries, which are maps from a set of keys to values. In the sequel, each key corresponds to a column and the mapped value corresponds to a valuation of the column in a particular row.

A dictionary `Dict <r> k v` maps a key `x` of type `k` to a value of type `v` that satisfies the property `r x`

```
type Range k v = k → v → Bool

data Dict k v <r :: Range k v> = D {
    dkeys :: [k]
  , dfun  :: x:{k | x ∈ elts dkeys} → v <r x>
  }
```

Each dictionary `d` has a domain `dkeys` *i.e.* the list of keys for which `d` is defined and a function `dfun` that is defined only on elements `x` of the domain `dkeys`. For each such element `x`, `dfun` returns a value that satisfies the property `r x`.

Propositions about the theory of sets can be decided efficiently by modern SMT solvers. Hence we use such propositions within refinements as demonstrated in chapter 1. The measures (logical functions) `elts` and `keys` specify the set of keys in a list and a dictionary respectively:

```
elts         :: [a] → Set a
elts ([])    = ∅
elts (x:xs) = {x} ∪ elts xs
```

```
keys         :: Dict k v → Set k
keys d       = elts (dkeys d)
```

Domain and Range of dictionaries. In order to precisely define the domain (*e.g.* columns) and range (*e.g.* values) of a dictionary (*e.g.* row), we define the following aliases:

```
type RD k v <dom :: Dom k v, rng :: Range k v>
  = {v:Dict <rng> k v | dom v}


type Dom k v = Dict k v → Bool
```

We may instantiate `dom` and `rng` with predicates that precisely describe the values contained with the dictionary. For example,

```
RD < \d → keys d = {"x"}, \k v→ 0 < v> String Int
```

describes dictionaries with a single field `"x"` whose value (as determined by `dfun`) is stricly greater than 0. We will define schemas by appropriately instantiating the abstract refinements `dom` and `rng`.

An empty dictionary has an empty domain and a function that will never be called:

```
empty   :: RD <emptyRD, rFalse> k v
empty   = D [] (\x → error "calling empty")


emptyRD = \d → keys d == ∅
rFalse  = \k v → false
```

We define singleton maps as dependent pairs `x := y` which denote the mapping from `x` to `y`:

```
data P k v <r :: Range k v>
  = (:=) {pk :: k, pv :: v <r pk>}
```

Thus, `key := val` has type `P <r> k v` only if `r key val`.

A dictionary may be extended with a singleton binding (which maps the new key to its new value).

```
(+=)   :: bind:P<r> k v
       → dict:RD<pTrue, r> k v
       → RD <addKey (pk bind) dict, r> k v

(k := v) += (D ks f)
       = D (k:ks) (\i → if i == k then v else f i)

addKey = \k d d′ → keys d′ == {k} ∪ keys d
pTrue  = \_ → True
```

Thus, `(k := v) += d` evaluates to a dictionary `d′` that extends `d` with the mapping from `k` to `v`. The type of `(+=)` constrains the new binding `bind`, the old dictionary `dict` and the returned value to have the same range invariant `r`. The return type states that the output dictionary's domain is that of the domain of `dict` extended by the new key `(pk bind)`.

To model a row in a table *i.e.* a schema, we define the unrefined (Haskell) type `Schema`, which is a dictionary mapping `Strings`, *i.e.* the names of the fields of the row, to elements of some universe `Univ` containing `Int`, `String` and `Double`. (A closed universe is not a practical restriction; most databases support a fixed set of types).

```
data Univ   = I Int | S String | D Double

type Schema = RD String Univ
```

We refine Schema with concrete instantiations for `dom` and `rng`, in order to recover precise specifications for a particular database. For example, `MovieSchema` is a refined `Schema` that describes the rows of the Movie table in Figure 4.1:

```
type MovieSchema = RD <md, mr> String Univ

md = \d → keys d={"year","star","dir","title"}
mr = \k v →
     (k = "year"  ⇒ isI v ∧ 1888 < toI v)
   ∧ (k = "star"  ⇒ isD v ∧ 0 ≤ toD v ≤ 10)
   ∧ (k = "dir"   ⇒ isS v)
   ∧ (k = "title" ⇒ isS v)
```

```
isI (I _)   = True
isI _       = False

toI       :: {v: Univ | isI v} → Int
toI (I n) = n
```

The predicate `md` describes the *domain* of the movie schema, restricting the keys to exactly `"year"`, `"star"`, `"dir"`, and `"title"`. The range predicate `mr` describes the types of the values in the schema: a dictionary of type `MovieSchema` must map `"year"` to an `Int`, `"star"` to a `Double`, and `"dir"` and `"title"` to `Strings`. The range predicate may be used to impose additional constraints on the values stored in the dictionary. For instance, `mr` restricts the year to be not only an integer but also greater than `1888`.

We populate the Movie Schema by extending the empty dictionary with the appropriate pairs of fields and values. For example, here are the rows from the table in Figure 4.1

```
movie1, movie2 :: MovieSchema
movie1 = ("title" := S "Persepolis")
       += ("dir"   := S "Paronnaud")
       += ("star"  := D 8)
       += ("year"  := I 2007)
       += empty

movie2 = ("title" := S "Birdman")
       += ("star"  := D 8.1)
       += ("dir"   := S "Inarritu")
       += ("year"  := I 2014)
       += empty
```

Typing `movie1` (and `movie2`) as `MovieSchema` boils down to proving that `keys movie1 = {"year", "star", "dir", "title"}` and that each key is mapped to an appropriate value as determined by `mr`. For example, declaring `movie1`'s year to be `I 1888` or even misspelling `"dir"` as `"Dir"` will cause the `movie1` to become ill-typed. As the (sub)typing relation depends on logical implication (unlike in `HList` based approaches [50]) key ordering *does not* affect

type-checking; in `movie1` the star field is added before the director, while `movie2` follows the opposite order.

Database Tables are collections of rows, *i.e.* collections of refined dictionaries. We define a type alias `RT s` (Refined Table) for the list of refined dictionaries from the field type `String` to the `Universe`.

```
type RT (s :: {dom::TDom, rng::TRange})
  = [RD <s.dom, s.rng> String Univ]

type TDom   = Dom   String Univ
type TRange = Range String Univ
```

For brevity we pack both the domain and the range refinements into a record `s` that describes the schema refinement; *i.e.* each row dictionary has domain `s.dom` and range `s.rng`.

For example, the table from Figure 4.1 can be represented as a type `MoviesTable` which is an `RT` refined with the domain and range `md` and `mr` described earlier (§ 4.3.1):

```
type MoviesTable = RT {dom = md, rng = mr}

movies :: MoviesTable
movies = [movie1, movie2]
```

4.3.2 Relational Algebra

Next, we describe the types of the relational algebra operators which can be used to manipulate refined rows and tables. For space reasons, we show the *types* of the basic relational operators; their (verified) implementations can be found online [90].

```
union   :: RT s → RT s → RT s
diff    :: RT s → RT s → RT s
select  :: (RD s → Bool) → RT s → RT s
project :: ks:[String] → RTSubEqFlds ks s
        → RTEqFlds ks s
product :: ( Disjoint s1 s2, Union s1 s2 s
           , Range s1 s, Range s2 s)
        ⇒ RT s1 → RT s2 → RT s
```

Union and diff compute the union and difference, respectively of the (rows of) two tables. The types of `union` and `diff` state that the operators work on tables with the same schema `s` and return a table with the same schema.

select takes a predicate `p` and a table `t` and filters the rows of `t` to those which that satisfy `p`. The type of `select` ensures that `p` will not reference columns (fields) that are not mapped in `t`, as the predicate `p` is constrained to require a dictionary with schema `s`.

project takes a list of `String` fields `ks` and a table `t` and projects exactly the fields `ks` at each row of `t`. `project`'s type states that for any schema `s`, the input table has type `RTSubEqFlds ks s` *i.e.* its domain should have at least the fields `ks` and the result table has type `RTEqFlds ks s`, *i.e.* its domain has exactly the elements `ks`.

```
type RTSubEqFlds ks s = RT s{dom = \z → elts ks ⊆  keys z}

type RTEqFlds ks s    = RT s{dom = \z → elts ks = keys z}
```

The range of the argument and the result tables is the same and equal to `s.rng`.

product takes two tables as input and returns their (Cartesian) product. It takes two Refined Tables with schemata `s1` and `s2` and returns a Refined Table with schema `s`. Intuitively, the output schema is the "concatenation" of the input schema; we formalize this notion using bounds: (1) `Disjoint s1 s2` says the domains of `s1` and `s2` should be disjoint, (2) `Union s1 s2 s` says the domain of `s` is the union of the domains of `s1` and `s2`, (3) `Range s1 s` (*resp.* `Range s2 s2`) says the range of `s1` should imply the result range `s`; together the two imply the output schema `s` preserves the type of each key in `s1` or `s2`.

```
bound Disjoint s1 s2 = \x y →
  s1.dom x ⇒ s2.dom y ⇒ keys x ∩ keys y == ∅

bound Union s1 s2 s = \x y v →
  s1.dom x ⇒ s2.dom y
           ⇒ keys v == keys x ∪ keys y
           ⇒ s.dom v
```

```
bound Range si s = \x k v →
  si.dom x ⇒ k ∈ keys x ⇒ si.rng k v ⇒ s.rng k v
```

Thus, bounded refinements enable the precise typing of relational algebra operations. They let us describe precisely when union, intersection, selection, projection and products can be computed, and let us determine, at compile time the exact "shape" of the resulting tables.

We can query Databases by writing functions that use the relational algebra combinators. For example, here is a query that returns the "good" titles – with more than 8 stars – from the `movies` table [1]

```
good_titles = project ["title"] $ select (\d →
                toDouble (dfun d $ "star") > 8
              ) movies
```

Finally, note that our entire library – including records, tables, and relational combinators – is built using vanilla Haskell *i.e.* without *any* type level computation. All schema reasoning happens at the granularity of the logical refinements. That is if the refinements are erased from the source, we still have a well-typed Haskell program but of course, lose the safety guarantees about operations (*e.g.* "dynamic" key lookup) never failing at run-time.

4.4 A Refined IO Monad

Next, we illustrate the expressiveness of Bounded Refinements by showing how they enable the specification and verification of stateful computations. We show how to (1) implement a refined *state transformer* (`RIO`) monad, where the transformer is indexed by refinements corresponding to *pre-* and *post*-conditions on the state (§ 4.4.1), (2) extend `RIO` with a set of combinators for *imperative* programming, *i.e.* whose types precisely encode Floyd-Hoare style program logics (§ 4.4.2), and (3) use the `RIO` monad to write *safe scripts* where the type system precisely tracks capabilities and statically ensures that functions only access specific resources (§ 4.5).

[1] More example queries can be found online [90]

4.4.1 The RIO Monad

The RIO data type describes stateful computations. Intuitively, a value of type `RIO a` denotes a computation that, when evaluated in an input `World` produces a value of type `a` (or diverges) and a potentially transformed output `World`. We implement `RIO a` as an abstractly refined type (as described in [98])

```
type Pre    = World → Bool
type Post a = World → a → World → Bool

data RIO a <p :: Pre, q :: Post a> = RIO {
  runState :: w:World <p> → (x:a, World <q w x>)
}
```

That is, `RIO a` is a function `World→ (a, World)`, where `World` is a primitive type that represents the state of the machine *i.e.* the console, file system, *etc.* This indexing notion is directly inspired by the method of [32] (also used in [68]).

Our Post-conditions are Two-State Predicates that relate the input- and output- world (as in [68]). Classical Floyd-Hoare logic, in contrast, uses assertions which are single-state predicates. We use two-states to smoothly account for specifications for stateful procedures. This increased expressiveness makes the types slightly more complex than a direct one-state encoding which is, of course also possible with bounded refinements.

An RIO computation is parameterized by two abstract refinements: (1) `p :: Pre`, which is a predicate over the *input* world, *i.e.* the input world `w` satisfies the refinement `p w`; and (2) `q :: Post a`, which is a predicate relating the *output* world with the input world and the value returned by the computation, *i.e.* the output world `w′` satisfies the refinement `q w x w′` where `x` is the value returned by the computation. Next, to use `RIO` as a monad, we define `bind` and `return` functions for it, that satisfy the monad laws.

The return operator yields a pair of the supplied value `z` and the input world unchanged:

```
return   :: z:a → RIO <p, ret z> a
return z = RIO $ \w → (z, w)
```

```
ret z    = \w x w' → w' == w ∧ x == z
```

The type of `return` states that for any precondition `p` and any supplied value `z` of type `a`, the expression `return z` is an `RIO` computation with precondition `p` and a post-condition `ret z`. The postcondition states that: (1) the output `World` is the same as the input, and (2) the result equals to the supplied value `z`. Note that as a consequence of the equality of the two worlds and congruence, the output world `w'` trivially satisfies `p w'`.

The bind Operator is defined in the usual way. However, to type it precisely, we require bounded refinements.

```
(>>=) :: (Ret q1 r, Seq r q1 p2, Trans q1 q2 q)
      ⇒ m:RIO <p, q1> a
      → k:(x:a<r> → RIO <p2 x, q2 x> b)
      → RIO <p, q> b

(RIO g) >>= f = RIO $ \x →
  case g x of { (y, s) → runState (f y) s }
```

The bounds capture various sequencing requirements (c.f. the Floyd-Hoare rules of consequence). First, the output of the first action `m`, satisfies the refinement required by the continuation `k`;

```
bound Ret q1 r = \w x w' → q1 w x w' ⇒ r x
```

Second, the computations may be sequenced, *i.e.* the postcondition of the first action `m` implies the precondition of the continuation `k` (which may be dependent upon the supplied value `x`):

```
bound Seq q1 p2 = \w x w' → q1 w x w' ⇒ p2 x w'
```

Third, the transitive composition of the two computations, implies the final postcondition:

```
bound Trans q1 q2 q = \w x w' y w'' →
      q1 w x w' ⇒ q2 x w' y w'' ⇒ q w y w''
```

Both type signatures would be impossible to use if the programmer had to manually instantiate the abstract refinements (*i.e.* pre- and post-conditions). Fortunately, Liquid Type inference generates the instantiations making it practical to use LIQUID HASKELL to verify stateful computations written using `do`-notation.

4.4.2 Floyd-Hoare Logic in the RIO Monad

Next, we use bounded refinements to derive an encoding of Floyd-Hoare logic, by showing how to read and write (mutable) variables and typing higher order `ifM` and `whileM` combinators.

We Encode Mutable Variables as fields of the `World` type. For example, we might encode a global counter as a field:

```
data World = { ... , ctr :: Int, ... }
```

We encode mutable variables in the refinement logic using McCarthy's `select` and `update` operators for finite maps and the associated axiom:

```
select :: Map k v → k → v
update :: Map k v → k → v → Map k v

∀ m, k1, k2, v.
     select (update m k1 v) k2
  == (if k1 == k2 then v else select m k2 v)
```

The quantifier free theory of `select` and `update` is decidable and implemented in modern SMT solvers [4].

We Read and Write Mutable Variables via suitable "get" and "set" actions. For example, we can read and write `ctr` via:

```
getCtr   :: RIO <pTrue, rdCtr> Int
getCtr   = RIO $ \w → (ctr w, w)

setCtr   :: Int → RIO <pTrue, wrCtr n> ()
setCtr n = RIO $ \w → ((), w { ctr = n })
```

Here, the refinements are defined as:

```
pTrue = \w → True
rdCtr = \w x w′ → w′ == w ∧ x == select w ctr
wrCtr n = \w _ w′ → w′ == update w ctr n
```

Hence, the post-condition of `getCtr` states that it returns the current value of `ctr`, encoded in the refinement logic with McCarthy's `select` operator while leaving the world unchanged. The post-condition of `setCtr` states that `World` is updated at the address corresponding to `ctr`, encoded via McCarthy's `update` operator.

The ifM combinator takes as input a `cond` action that returns a `Bool` and, depending upon the result, executes either the `then` or `else` actions. We type it as:

```
bound Pure g    = \w x v → (g w x v ⇒ v == w)
bound Then g p1 = \w v   → (g w True  v ⇒ p1 v)
bound Else g p2 = \w v   → (g w False v ⇒ p2 v)

ifM :: (Pure g, Then g p1, Else g p2)
    ⇒ RIO <p , g> Bool       -- cond
    → RIO <p1, q> a          -- then
    → RIO <p2, q> a          -- else
    → RIO <p , q> a
```

The abstract refinements and bounds correspond exactly to the hypotheses in the Floyd-Hoare rule for the `if` statement. The bound `Pure g` states that the `cond` action may access but does not *modify* the `World`, *i.e.* the output is the same as the input `World`. (In classical Floyd-Hoare formulations this is done by syntactically separating terms into pure *expressions* and side effecting *statements*). The bound `Then g p1` and `Else g p2` respectively state that the preconditions of the `then` and `else` actions are established when the `cond` returns `True` and `False` respectively.

We can use ifM to implement a stateful computation that performs a division, after checking the divisor is non-zero. We specify that `div` should not be called with a zero divisor. Then, LIQUID HASKELL verifies that `div` is called safely:

```
div :: Int → {v:Int | v ≠ 0} → Int

ifTest :: RIO Int
ifTest = ifM nonZero divX (return 10)
  where nonZero = getCtr >>= return . (≠ 0)
        divX    = getCtr >>= return . (div 42)
```

Verification succeeds as the post-condition of `nonZero` is instantiated to `\_ b w` $\rightarrow$ `b` $\Leftrightarrow$ `select w ctr` $\neq$ `0` and the pre-condition of `divX`'s is instantiated to `\w` $\rightarrow$ `select w ctr` $\neq$ `0`, which suffices to prove that `div` is only called with non-zero values.

The whileM combinator formalizes loops as `RIO` computations:

```
whileM :: (OneState q, Inv p g b, Exit p g q)
       ⇒ RIO <p, g> Bool      -- cond
       → RIO <pTrue, b> ()    -- body
       → RIO <p, q> ()
```

As with `ifM`, the hypotheses of the Floyd-Hoare derivation rule become bounds for the signature. Given a `cond`ition with pre-condition `p` and post-condition `g` and `body` with a true precondition and post-condition `b`, the computation `whileM cond body` has precondition `p` and post-condition `q` as long as the bounds (corresponding to the Hypotheses in the Floyd-Hoare derivation rule) hold. First, `p` should be a loop invariant; *i.e.* when the `cond`ition returns `True` the post-condition of the body `b` must imply the `p`:

```
bound Inv p g b = \w w′ w′′ →
    p w ⇒ g w True w′ ⇒ b w′ () w′′ ⇒ p w′′
```

Second, when the `cond`ition returns `False` the invariant `p` should imply the loop's post-condition `q`:

```
bound Exit p g q = \w w′ →
      p w ⇒ g w False w′ ⇒ q w () w′
```

Third, to avoid having to transitively connect the guard and the body, we require that the loop post-condition be a one-state predicate, independent of the input world (as in Floyd-Hoare logic):

```
bound OneState q = \w w′ w′′ →
      q w () w′′ ⇒ q w′ () w′′
```

We can use whileM to implement a loop that repeatedly decrements a counter while it is positive, and to then verify that if it was initially non-negative, then at the end the counter is equal to `0`.

```
whileTest   :: RIO <posCtr, zeroCtr> ()
whileTest = whileM gtZeroX decr
```

```
  where gtZeroX = getCtr >>= return . (> 0)


posCtr  = \w → 0 ≤ select w ctr
zeroCtr = \_ _ w′ → 0 == select w ctr
```

Where the decrement is implemented by `decr` with type:

```
decr :: RIO <pTrue, decCtr> ()


decCtr = \w _ w′ → w′ == update w ctr ((select ctr w) - 1)
```

LIQUID HASKELL verifies that at the end of `whileTest` the counter is zero (*i.e.* the post-condition `zeroCtr`) by instantiating suitable (*i.e.* inductive) refinements for this particular use of `whileM`.

4.5 Capability Safe Scripting via RIO

```
pread, pwrite, plookup, pcontents,
pcreateD, pcreateF, pcreateFP :: Priv → Bool

active   :: World → Set FH
caps     :: World → Map FH Priv

pset p h = \w → p (select (caps w) h) ∧ h ∈ active w
```

Figure 4.4. Privilege Specification.

Next, we describe how we use the `RIO` monad to reason about shell scripting, inspired by the `Shill` [66] programming language.

Shill is a scripting language that restricts the privileges with which a script may execute by using *capabilities* and *dynamic contract checking* [66] . Capabilities are *run-time values* that witness the right to use a particular resource (*e.g.* a file). A capability is associated with a set of privileges, each denoting the permission to use the capability in a particular way (such as the permission to write to a file). A contract for a `Shill` procedure describes the required input capabilities and any output values. The `Shill` runtime guarantees that system resources are accessed in the manner described by its contract.

In this section, we turn to the problem of preventing `Shill` runtime failures. (In general,

the verification of file system resource usage is a rich topic outside the scope of this paper). That is, assuming the `Shill` runtime and an API as described in [66], how can we use Bounded Refinement Types to encode scripting privileges and reason about them *statically?*

We use RIO types to specify `Shill`'s API operations thereby providing *compile-time* guarantees about privilege and resource usage. To achieve this, we: connect the state (`World`) of the `RIO` monad with a *privilege specification* denoting the set of privileges that a program may use (§ 4.5.1); specify the *file system API* in terms of this abstraction (§ 4.5.2); and use the above to specify and verify the particular privileges that a *client* of the API uses (§ 4.5.3).

4.5.1 Privilege Specification

Figure 4.4 summarizes how we specify privileges inside `RIO`. We use the type `FH` to denote a file handles, analogous to `Shill`'s capabilities. An abstract type `Priv` denotes the sets of privileges that may be associated with a particular `FH`.

To connect Worlds with Privileges we assume a set of uninterpreted functions of type `Priv` $\rightarrow$ `Bool` that act as predicates on values of type `Priv`, each denoting a particular privilege. For example, given a value `p :: Priv`, the proposition `pread p` denotes that `p` includes the "read" privilege. The function `caps` associates each `World` with a `Map FH Priv`, a table that associates each `FH` with its privileges. The function `active` maps each `World` to the `Set` of allocated `FH`s. Given `x:FH` and `w:World`, `pwrite (select (caps w) x)` denotes that in the state `w`, the file `x` may be written. This pattern is generalized by the predicate `pset pwrite x w`.

4.5.2 File System API Specification

A privilege tracking file system API can be partitioned into the privilege *preserving* operations and the privilege *extending* operations.

To type the privilege preserving operations, we define a predicate `eqP w w'` that says that the set of privileges and active handles in worlds `w` and `w'` are *equivalent*.

```
eqP = \w _ w' → caps w == caps w' ∧ active w == active w'
```

We can now specify the privilege preserving operations that `read` and `write` files, and list the `contents` of a directory, all of which require the capabilities to do so in their pre-conditions:

```
read :: {- Read the contents of h -}
  h:FH → RIO <pset pread h, eqp> String

write :: {- Write to the file h -}
  h:FH → String → RIO <pset pwrite h, eqp> ()

contents :: {- List the children of h -}
  h:FH → RIO <pset pcontents h, eqp> [Path]
```

To type the privilege extending operations, we define predicates that say that the output world is suitably extended. First, each such operation *allocates* a new handle, which is formalized as:

```
alloc w′ w x = (x ∉ active w) ∧ active w′ == {x} ∪ active w
```

which says that the active handles in (the new `World`) `w′` are those of (the old `World`) `w` extended with the hitherto *inactive* handle `x`. Typically, after allocating a new handle, a script will want to add privileges to the handle that are obtained from existing privileges.

To create a new file in a directory with handle `h` we want the new file to have the privileges *derived* from `pcreateFP (select (caps w) h)` (*i.e.* the create privileges of `h`). We formalize this by defining the post-condition of `create` as the predicate `derivP`:

```
derivP h  = \w x w′ →
  alloc w′ w x ∧
  caps w′ == store (caps w) x (pcreateFP (select (caps w)) h)

create :: {- Create a file -}
  h:FH → Path → RIO <pset pcreateF h, derivP h> FH
```

Thus, if `h` is writable in the old `World w` (`pwrite (pcreateFP (select (caps w) h))`) and `x` is derived from `h` (`derivP w′ w x h` both hold), then we know that `x` is writable in the new `World w′` (`pwrite (select (caps w′) x)`).

To lookup existing files or create sub-directories, we want to directly *copy* the privileges of the parent handle. We do this by using a predicate `copyP` as the post-condition for the two functions:

```
copyP h = \w x w' →
  alloc w' w x ∧
  caps w' == store (caps w) x
                   (select (caps w) y)

lookup :: {- Open a child of h -}
  h:FH→Path→RIO<pset plookup h, copyP h> FH

createDir :: {- Create a directory -}
  h:FH→Path→RIO<pset pcreateD h, copyP h> FH
```

4.5.3 Client Script Verification

We now turn to a client script, the program `copyRec` that copies the contents of the directory `f` to the directory `d`.

```
copyRec recur s d =
  do cs <- contents s
     forM_ cs $ \ p → do
       x <- flookup s p
       when (isFile x) $ do
         y <- create d p
         s <- fread x
         write y s
       when (recur ∧ (isDir x)) $ do
         y <- createDir d p
         copyRec recur x y
```

`copyRec` executes by first listing the contents of `f`, and then opening each child path `p` in `f`. If the result is a file, it is copied to the directory `d`. Otherwise, `copyRec` recurses on `p`, if `recur` is true.

In a first attempt to type `copyRec` we give it the following type:

```
copyRec :: Bool → s:FH → d:FH →
           RIO<copySpec s d, \_ _ w → copySpec s d w> ()

copySpec h d = \w →
```

```
pset pcontents h w ∧ pset plookup h     w ∧
pset pread h     w ∧ pset pcreateFile d w ∧
pset pwrite d    w ∧ pset pcreateF d    w ∧
pwrite (pcreateFP (select (caps w) d)))
```

The above specification gives `copyRec` a minimal set of privileges. Given a source directory handle `s` and destination handle `d`, the `copyRec` must at least: (1) list the contents of `s` (`pcontents`), (2) open children of `s` (`plookup`), (3) read from children of `s` (`pread`), (4) create directories in `d` (`pcreateD`), (5) create files in `d` (`pcreateF`), an (6) write to (created) files in `d` (`pwrite`). Furthermore, we want to restrict the privileges on newly created files to the write privilege, since `copyRec` does not need to read from or otherwise modify these files.

Even though the above type is sufficient to verify the various clients of `copySpec` it is insufficient to verify `copySpec`'s implementation, as the postcondition merely states that `copySpec s d w` holds. Looking at the recursive call in the last line of `copySpec`'s implementation, the output world `w` is only known to satisfy `copySpec x y w` (having substituted the formal parameters `s` and `d` with the actual `x` and `y`), with no mention of `s` or `d`! Thus, it is impossible to satisfy the postcondition of `copyRec`, as information about `s` and `d` has been lost.

Framing is introduced to address the above problem. Intuitively, because no privileges are ever *revoked*, if a privilege for a file existed *before* the recursive call, then it exists *after* as well. We thus introduce a notion of *framing* – assertions about unmodified state that hold before calling `copyRec` must hold after `copyRec` returns. Solidifying this intuition, we define a predicate `i` to be `Stable` when assuming that the predicate `i` holds on `w`, if `i` only depends on the allocated set of privileges, then `i` will hold on a world `w′` so long as the set of priviliges in `w′` contains those in `w`. The definition of `Stable` is derived precisely from the ways in which the file system API may modify the current set of privileges:

```
bound Stable i = \x y w w′ →
 i w ⇒ ( eqP w () w′ || copyP y w x w′ || derivP y w x w′)
     ⇒ i w′
```

We thus parameterize `copyRec` by a predicate `i`, bounded by `Stable i`, which precisely describes the possible world transformations under which `i` should be stable:

```
copyFrame i s d = \w → i w ∧ copySpec s d w


copyRec :: (Stable i)
        ⇒ Bool → s:FH → d:FH
        → RIO<copyFrame i s d, \_ _ w → copyFrame i s d w> ()
```

Now, we can verify `copyRec`'s body, as at the recursive call that appears in the last line of the implementation, `i` is instantiated with `\w → copySpec s d w`.

4.6 Conclusion

We presented a notion of bounded quantification for refinement types and show how it expands the expressiveness of refinement typing by using it to develop typed combinators for: (1) relational algebra and safe database access, (2) Floyd-Hoare logic within a state transformer monad equipped with combinators for branching and looping, and (3) using the above to implement a refined IO monad that tracks capabilities and resource usage. This leap in expressiveness comes via a translation to "ghost" functions, which lets us retain the automated and decidable SMT based checking and inference that makes refinement typing effective in practice.

With Bounded Refinement Types we get *relatively complete* expressiveness of specification, that is we can use refinement types to express any property referring to the underlying (decidable) logic. Thus our expressiveness power is still limited, we cannot use arbitrary Haskell functions is the specifications. Next we see how to overcome this expressiveness limitation and thus allow arbitrary Haskell expressions into the specifications while still preserving decidable type checking.

Acknowledgments The material of this chapter are adapted from the following publication: N. Vazou, A. Bakst, and R. Jhala, "Bounded Refinement Types", ICFP, 2015.

Chapter 5

Refinement Reflection

> Did you ever wonder if the person in the puddle is real,
> and you're just a reflection of him?
>
> *– Bill Watterson*

In this chapter we introduce *Refinement Reflection*, a method to extend *legacy* languages—with highly tuned libraries, compilers, and run-times—into theorem provers, by letting programmers specify and verify arbitrary properties of their code simply by writing programs in the legacy language.

Refinement types, as presented so far, offer a form of programming with proofs that can be retrofitted into a programming language. The retrofitting relies upon restricting refinements to so-called "shallow" specifications that correspond to *abstract* interpretations of the behavior of functions. For example, refinements make it easy to specify that the list returned by the `append` function has size equal to the sum of those of its inputs. These shallow specifications fall within decidable logical fragments, and hence, can be automatically verified using SMT based refinement typing.

Refinements are a pale shadow of what is possible with dependently typed languages like Coq, Agda and Idris which permit "deep" specification and verification. These languages come equipped with mechanisms that *represent* and *manipulate* the exact descriptions of user-defined functions. For example, we can represent the specification that the `append` function is associative, and we can manipulate (unfold) its definition to write a small program that constructs a proof

of the specification. Dafny [54], F* [88] and Halo [103] take a step towards SMT-based deep verification, by encoding user-defined functions as universally quantified logical formulas or "axioms". This axiomatic approach offers significant automation but relies heavily upon brittle heuristics for "triggering" axiom instantiation, giving away decidable, and hence, predictable verification [56].

In this chapter, we present a new approach to retrofitting deep verification into existing languages. Our approach reconciles the automation of SMT-based refinement typing with decidable and predictable verification, and enables users to reify pencil-and-paper proofs simply as programs in the source language.

- We start this chapter by an overview of refinement reflection: the code implementing a user-defined function can be *reflected* into the function's (output) refinement type, thus converting the function's (refinement) type signature into a deep specification of the functions behavior. This simple idea has a profound consequence: at *uses* of the function, the standard rule for (dependent) function application yields a precise, predictable and most importantly, programmer controllable means of *instantiating* the deep specification that is not tethered to brittle SMT heuristics. Specifically, we show how to use ideas for *defunctionalization* from the theorem proving literature which encode functions and lambdas using uninterpreted symbols, to encode terms from an expressive higher order language as decidable refinements, letting us use SMT-based congruence closure for decidable and predictable verification (§ 5.2).

- Next, we present a *library of combinators* that lets programmers *compose proofs* from basic refinements and function definitions. We show how to represent proofs simply as unit-values refined by the proposition that they prove. We show how to build up sophisticated proofs using a small library of combinators that permits reasoning in an algebraic or equational style. Furthermore, since proofs are literally just programs, our proof combinators let us use standard language mechanisms like branches (to encode case splits), recursion (to encode induction), and functions (to encode auxiliary lemmas) to write proofs that look very much like transcriptions of their pencil-and-paper analogues (§ 5.1).

- We implemented refinement reflection in LIQUID HASKELL [100], thereby converting the legacy language Haskell into a theorem prover. We demonstrate the benefits of this conversion by proving typeclass laws. Haskell's typeclass machinery has led to a suite of expressive abstractions and optimizations which, for correctness, crucially require typeclass *instances* to obey key algebraic laws. We show how reflection can be used to formally verify that many widely used instances of the Monoid, Applicative, Functor, and Monad typeclasses actually satisfy the respective laws, making the use of these typeclasses safe (§ 5.4).

- Finally, to showcase the benefits of retrofitting theorem proving onto legacy languages, we perform a case study in *deterministic parallelism*. Existing deterministic languages place unchecked obligations on the user to guarantee, *e.g.* the associativity of a fold. Violations can compromise type soundness and correctness. Closing this gap requires only modest proof effort—touching only a small subset of the application. But for this solution to be possible requires a *practical*, *parallel* programming language that supports deep verification. Before LIQUID HASKELL there was no such parallel language. We show how LIQUID HASKELL lets us verify the unchecked obligations from benchmarks taken from three existing parallel programming systems, and thus, paves the way towards high-performance with correctness guarantees (§ 5.5).

5.1 Overview

We begin with an overview of refinement reflection and how it allows us to write proofs *of* and *by* Haskell functions.

5.1.1 Refinement Types

First, we recall some preliminaries about refinement types and how they enable shallow specification and verification.

Refinement types are the source program's (here Haskell's) types decorated with logical predicates drawn from a(n SMT decidable) logic [20, 81]. For example, we can define the `Nat` type by

refining Haskell's `Int` type with a predicate `0 ≤ v`:

```
type Nat = { v:Int | 0 ≤ v }
```

Here, `v` names the value described by the type: the above can be read as the "set of `Int` values `v` that are not less than 0". The refinement is drawn from the logic of quantifier free linear arithmetic and uninterpreted functions (QF-UFLIA [4]).

Specification & Verification We can use refinements to define and type the textbook Fibonacci function as:

```
fib :: Nat → Nat
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Here, the input type's refinement specifies a *pre-condition* that the parameters must be `Nat`, which is needed to ensure termination, and the output types's refinement specifies a *post-condition* that the result is also a `Nat`. Refinement type checking lets us specify and (automatically) verify the shallow property that if `fib` is invoked with a non-negative `Int`, then it terminates and yields a non-negative `Int`.

Propositions We can use refinements to define a data type representing propositions simply as an alias for unit, a data type that carries no useful runtime information:

```
type Prop = ()
```

which can be *refined* with propositions about the code. For example, the following states the proposition $2+2$ equals 4.

```
type Plus_2_2_eq_4 = { v: Prop | 2 + 2 = 4 }
```

For clarity, we abbreviate the above type by omitting the irrelevant basic type `Prop` and variable `v`:

```
type Plus_2_2_eq_4 = { 2 + 2 = 4 }
```

Function types encode universally quantified propositions:

```
type Plus_com = x:Int → y:Int → { x + y = y + x }
```

The parameters `x` and `y` refer to input values. Any inhabitant of the above type is a proof that `Int` addition is commutative.

Proofs We *prove* the above theorems by providing inhabitants to type specifications in forms of Haskell programs. To ease this task LIQUID HASKELL provides primitives to construct proof terms by "casting" expressions to `Prop`.

```
data QED = QED


(**) :: a → QED → Prop
_ ** _  = ()
```

To resemble mathematical proofs, we make this casting post-fix. Thus, we write `e ** QED` to cast `e` to a value of `Prop`. For example, we can prove the above propositions by writing

```
pf_plus_2_2 :: Plus_2_2_eq_4
pf_plus_2_2 = trivial ** QED


pf_plus_comm :: Plus_comm
pf_plus_comm = \x y → trivial ** QED


trivial = ()
```

Via standard refinement type checking, the above code yields the respective verification conditions (VCs),

$$2+2=4$$

$$\forall\, x,\, y\, .\, x+y=y+x$$

which are easily proved valid by the SMT solver, allowing us to prove the respective propositions.

A Note on Bottom: Readers familiar with Haskell's semantics may be feeling anxious about whether the dreaded "bottom", which inhabits all types, makes our proofs suspect. Fortunately, as described in [100], LIQUID HASKELL ensures that all terms with non-trivial refinements provably evaluate to (non-bottom) values, thereby making our proofs sound.

5.1.2 Refinement Reflection

Suppose we wish to prove properties about the `fib` function, *e.g.* `fib 2` equals `1`.

```
type fib2_eq_1 = { fib 2 = 1 }
```

Standard refinement type checking runs into two problems. First, for decidability and soundness, arbitrary user-defined functions do not belong the refinement logic, *i.e.* we cannot *refer* to `fib` in a refinement. Second, the only information that a refinement type checker has about the behavior of `fib` is its shallow type specification `Nat` $\rightarrow$ `Nat` which is far too weak to verify `fib2_eq_1`. To address both problems, we use the following annotation, which sets in motion the three steps of refinement reflection:

```
reflect fib
```

Step 1: Definition The annotation tells LIQUID HASKELL to declare an *uninterpreted function* `fib :: Int` $\rightarrow$ `Int` in the refinement logic. By uninterpreted, we mean that the logical `fib` is *not* connected to the program function `fib`; in the logic, `fib` only satisfies the *congruence axiom*

$$\forall n, m.\ n = m \ \Rightarrow\ \texttt{fib}\ n = \texttt{fib}\ m$$

On its own, the uninterpreted function is not terribly useful, as it does not let us prove `fib2_eq_1` which requires reasoning about the *definition* of `fib`.

Step 2: Reflection In the next key step, LIQUID HASKELL reflects the definition into the refinement type of `fib` by automatically strengthening the user defined type for `fib` to:

```
fib :: n:Nat → { v:Nat | fibP v n }
```

where `fibP` is an alias for a refinement *automatically derived* from the function's definition:

```
fibP v n = v = if n = 0 then 0 else
               if n = 1 then 1 else
               fib(n-1) + fib(n-2)
```

Step 3: Application With the reflected refinement type, each application of `fib` in the code automatically unfolds the `fib` definition *once* in the logic. We prove `fib2_eq_1` by:

```
pf_fib2 :: { fib 2 = 1 }
pf_fib2 = let t0 = fib 0
              t1 = fib 1
              t2 = fib 2
          in  ()
```

We write `f` to denote places where the unfolding of `f`'s definition is important. Via refinement typing, the above proof yields the following verification condition that is discharged by the SMT solver, even though `fib` is uninterpreted:

$$(\texttt{fibP}\ (\texttt{fib}\ 0)\ 0) \wedge (\texttt{fibP}\ (\texttt{fib}\ 1)\ 1) \wedge (\texttt{fibP}\ (\texttt{fib}\ 2)\ 2) \Rightarrow (\texttt{fib}\ 2 = 1)$$

Note that the verification of `pf_fib2` relies merely on the fact that `fib` was applied to (*i.e.* unfolded at) `0`, `1` and `2`. The SMT solver automatically *combines* the facts, once they are in the antecedent. The following is also verified:

```
pf_fib2' :: { fib 2 = 1 }
pf_fib2' = [ fib 0, fib 1, fib 2 ] ** QED
```

Thus, unlike classical dependent typing, refinement reflection *does not* perform any type-level computation.

Reflection vs. Axiomatization An alternative *axiomatic* approach, used by Dafny [54] and F* [88], is to encode `fib` using a universally quantified SMT formula (or axiom):

$$\forall n.\ \texttt{fibP}\ (\texttt{fib}\ n)\ n$$

Axiomatization offers greater automation than reflection. Unlike LIQUID HASKELL, Dafny will verify the following by *automatically instantiating* the above axiom at `2`, `1` and `0`:

```
axPf_fib2 :: { fib 2 = 1 }
axPf_fib2 = trivial ** QED
```

The automation offered by axioms is a bit of a devil's bargain, as axioms render checking of the VCs *undecidable*. In practice, automatic axiom instantation can easily lead to infinite

"matching loops". For example, the existence of a term `fib` n in a VC can trigger the above axiom, which may then produce the terms `fib` $(n-1)$ and `fib` $(n-2)$, which may then recursively give rise to further instantiations *ad infinitum*. To prevent matching loops an expert must carefully craft "triggers" and provide a "fuel" parameter [1] that can be used to restrict the numbers of the SMT unfoldings, which ensure termination, but can cause the axiom to not be instantiated at the right places. In short, per the authors of Dafny, the undecidability of the VC checking and its attendant heuristics makes verification unpredictable [56].

5.1.3 Structuring Proofs

In contrast to the axiomatic approach, with refinement reflection, the VCs are deliberately designed to always fall in an SMT-decidable logic, as function symbols are uninterpreted. It is up to the programmer to unfold the definitions at the appropriate places, which we have found, with careful design of proof combinators, to be quite a natural and pleasant experience. To this end, we have developed a library of proof combinators that permits reasoning about equalities and linear arithmetic, inspired by Agda [67].

"Equation" Combinators We equip LIQUID HASKELL with a family of equation combinators `op.` for each logical operator `op` in $\{=, \neq, \leq, <, \geq, >\}$, the operators in the theory QF-UFLIA. The refinement type of `op.` *requires* that $x \odot y$ holds and then *ensures* that the returned value is equal to `x`. For example, we define `=.` as:

```
(=.) :: x:a → y:{a| x=y} → {v:a| v=x}
x =. _ = x
```

and use it to write the following "equational" proof:

```
eqPf_fib2 :: { fib 2 = 1 }
eqPf_fib2 =  fib 2
          =. fib 1 + fib 0
          =. 1
          ** QED
```

"Because" Combinators Often, we need to compose "lemmata" into larger theorems. For example, to prove `fib 3 = 2` we may wish to reuse `eqPf_fib2` as a lemma. To this end, LIQUID

HASKELL has a "because" combinator:

```
(∵) :: (Prop → a) → Prop → a
f ∵ y = f y
```

The operator is simply an alias for function application that lets us write `x op. y ∵ p` (instead of `(op.) x y p`) where `(op.)` is extended to accept an *optional* third proof argument via Haskell's typeclass mechanisms. We use the because combinator to prove that `fib 3 = 2` with a Haskell function:

```
eqPf_fib3 :: { fib 3 = 2 }
eqPf_fib3 =  fib 3
          =. fib 2 + fib 1
          =. 2              ∵ eqPf_fib2
          ** QED
```

Arithmetic and Ordering SMT based refinements let us go well beyond just equational reasoning. Next, lets see how we can use arithmetic and ordering to prove that `fib` is (locally) increasing, *i.e.* for all n, `fib` $n \leq$ `fib` $(n+1)$

```
fibUp :: n:Nat → { fib n ≤ fib (n+1) }
fibUp n
  | n == 0
  =  fib 0 <. fib 1
  ** QED


  | n == 1
  =  fib 1 ≤. fib 1 + fib 0 ≤. fib 2
  ** QED


  | otherwise
  =  fib n
  =. fib (n-1) + fib (n-2)
  ≤. fib n     + fib (n-2) ∵ fibUp (n-1)
  ≤. fib n     + fib (n-1) ∵ fibUp (n-2)
  ≤. fib (n+1)
  ** QED
```

Case Splitting and Induction The proof `fibUp` works by induction on `n`. In the *base* cases `0` and `1`, we simply assert the relevant inequalities. These are verified as the reflected refinement unfolds the definition of `fib` at those inputs. The derived VCs are (automatically) proved as the SMT solver concludes $0 < 1$ and $1 + 0 \leq 1$ respectively. In the *inductive* case, `fib n` is unfolded to `fib (n-1) + fib (n-2)`, which, because of the induction hypothesis (applied by invoking `fibUp` at `n-1` and `n-2`) and the SMT solver's arithmetic reasoning, completes the proof.

Higher Order Theorems Refinements smoothly accomodate higher-order reasoning. For example, lets prove that every locally increasing function is monotonic, *i.e.* if `f z ≤ f (z+1)` for all `z`, then `f x ≤ f y` for all `x < y`.

```
fMono :: f:(Nat → Int)
      → fUp:(z:Nat → {f z ≤ f (z+1)})
      → x:Nat
      → y:{x < y}
      → {f x ≤ f y} / [y]
fMono f inc x y
  | x + 1 == y
  =  f x ≤. f (x+1) ∵ fUp x
         ≤. f y
         ** QED


  | x + 1 < y
  =  f x ≤. f (y-1) ∵ fMono f fUp x (y-1)
         ≤. f y     ∵ fUp (y-1)
         ** QED
```

We prove the theorem by induction on `y`, which is specified by the annotation `/ [y]` which states that `y` is a well-founded termination metric that decreases at each recursive call [100]. If `x+1 == y`, then we use `fUp x`. Otherwise, `x+1 < y`, and we use the induction hypothesis *i.e.* apply `fMono` at `y-1`, after which transitivity of the less-than ordering finishes the proof. We can use the general `fMono` theorem to prove that `fib` increases monotonically:

```
fibMono :: n:Nat → m:{n<m} → {fib n ≤ fib m}
fibMono = fMono fib fibUp
```

5.1.4 Case Study: Deterministic Parallelism

One benefit of an in-language prover is that it lowers the barrier to *small* verification efforts that touch only a fraction of the program, and yet ensure critical invariants that Haskell's type system cannot. Here we consider parallel programming, which is commonly considered error prone and entails proof obligations on the user that typically go unchecked.

The situation is especially precarious with parallel programming frameworks that claim to be *determinstic* and thus usable within purely functional programs. These include Deterministic Parallel Java (DPJ [10]), Concurrent Revisions for .NET [14], and Haskell's LVish [52], Accelerate [64], and REPA [48]. Accelerate's parallel fold function, for instance, claims to be deterministic—and its purely functional type means the Haskell optimizer will *assume* its referential transparency—but its determinism depends on an associativity guarantee which must be assured *by the programmer* rather than the type system. Thus simply folding the minus function, `fold (-) 0 arr`, is sufficient to violate determinism and Haskell's pure semantics.

Likewise, DPJ goes to pains to develop a new type system for parallel programming, but then provides a "commutes" annotation for methods updating shared state, compromising the *guarantee* and going back to trusting the user. LVish has the same Achilles heel. Consider set insertion:

```
insert :: Ord a ⇒ a → Set s a → Par s ()
```

Here `insert` returns an (effectful) `Par` computation, which can be run within a pure function to produce a pure result. At first glance it would seem that trusting the implementation of the concurrent set is sufficient to assure a deterministic outcome. Yet the interface has an `Ord` constraint. This polymorphic function works with user-defined data types, and thus user-defined orderings. What if the user fails to implement a total order? Then, even a correct implementation of, e.g. a concurrent skiplist [34], can reveal different insertion orders due to concurrency.

In summary, parallel programs naturally need to communicate, but the mechanisms of that communication—such as folds or inserts into a shared structure—typically carry additional proof obligations. This in turn makes parallelism a liability. But we can remove the risk with verification.

Verified typeclasses Our solution is simply to change the `Ord` constraint above to `VerifiedOrd`.

```
insert :: VerifiedOrd a ⇒ a → Set s a → Par s ()
```

This constraint changes the interface but not the implementation of `insert`. The additional methods of the verified type class don't add operational capabilities, but rather impose additional proof obligations:

```
class Ord a ⇒ VerifiedOrd a where
 antisym :: x:a → y:a → { x ≤ y ∧ y ≤ x ⇒ x = y }
 trans   :: x:a → y:a → z:a → { x ≤ y ∧ y ≤ z ⇒ x ≤ z }
 total   :: x:a → y:a → { x ≤ y || y ≤ x }
```

Similarly, we can extend the `Monoid` typeclass to a `VerifiedMonoid`, with refinements expressing `Monoid` laws.

```
class Monoid a ⇒ VerifiedMonoid a where
 lident :: x:a → { ε ◇ x = x }
 rident :: x:a → { x ◇ ε = x }
 assoc  :: x:a → y:a → z:a → { x ◇ (y ◇ z) = (x ◇ y) ◇ z }
```

The `VerifiedMonoid` typeclass constraint requires the binary operation to be associative, thus can be safely used to fold on an unknown number of processors.

Verified instances for primitive types `VerifiedOrd` instances for primitive types like `Int`, `Double` are trivial to write; they just appeal to the SMT solver's built-in theories. For example, the following is a valid totality proof on `Int`.

```
totInt :: x:Int → y:Int → {x ≤ y || y ≤ x}
totInt _ _ = trivial ** QED
```

Verified instances for algebraic datatypes To prove the class laws for user defined algebraic datatypes, refinement reflection allows for structurally inductive proof terms. For example, we can inductively define Peano numerals

```
data Peano = Z | S Peano
```

We can compare two `Peano` numbers via

```
reflect leq :: Peano → Peano → Bool
leq Z _          = True
leq (S n) Z      = False
leq (S n) (S m) = leq n m
```

In § 5.2 we will describe exactly how the reflection mechanism (illustrated via `fibP`) is extended to account for ADTs like `Peano`. LIQUID HASKELL automatically checks that `leq` is total [100], which lets us safely `reflect` it into the logic.

Next, we prove that `leq` is total on `Peano` numbers

```
totalPeano :: n:Peano → m:Peano → {leq n m || leq m n} / [toInt n + toInt m]
totalPeano Z m = leq Z m ** QED
totalPeano n Z = leq Z n ** QED
totalPeano (S n) (S m)
 =  leq (S n) (S m) || leq (S m) (S n)
 =. leq n m || leq m n
 =. True ∵ totalPeano m n
 ** QED
```

The proof goes by induction, splitting cases on whether the number is zero or non-zero. Consequently, we pattern match on the parameters `n` and `m`, and furnish separate proofs for each case. In the "zero" cases, we simply unfold the definition of `leq`. In the "successor" case, after unfolding we (literally) apply the induction hypothesis by using the because operator. The termination hint `[toInt n + toInt m]`, where `toInt` maps `Peano` numbers to integers, is used to verify well-formedness of the `totalPeano` proof term. LIQUID HASKELL's termination and totality checker use the hint to verify that we are in fact doing induction properly (§ 5.2).

Similarly to `totalPeano`, we can define the rest of the `VerifiedOrd` proof methods and use them to create the verified instance.

```
instance Ord Peano where
  (≤) = leq

instance VerifiedOrd Peano where
  total = totalPeano
```

Proving all the four `VerifiedOrd` laws is a burden on the programmer. Since `Peano` is isomorphic to `Nats`, next we present how to reduce the `Peano` proofs into the SMT automated integer proofs.

Isomorphisms In order to reuse proofs for a custom datatype, we provide a way to translate verified instances between isomorphic types [5]. We design a typeclass `Iso` which witnesses the fact that two types are isomorphic.

```
class Iso a b where
  to      :: a → b
  from    :: b → a
  to∘from :: x:a → {to (from x) = x}
  from∘to :: x:a → {from (to x) = x}
```

For two isomorphic types `a` and `b` we compare instances of `b` using `a`'s comparison method.

```
instance (Ord a, Iso a b) ⇒ Ord b where
  x ≤ y = from x ≤ from y
```

Then, we prove that `VerifiedOrd` laws are closed under isomorphisms. For example, we prove totality of comparison on `b`s using the `VerifiedOrd` totality on `a`s

```
isoTotal :: (VerifiedOrd a, Iso a b) ⇒ x:b → y:b → {x ≤ y || y ≤ x}
isoTotal x y
 =  x ≤ y || y ≤ x
 =. (from x) ≤ (from y) || (from y) ≤ (from x)
    ∵ total (from x) (from y)
 ** QED
```

We use `isoTotal` to create a verified instance on `b`s.

```
instance (VerifiedOrd a, Iso a b) ⇒ VerifiedOrd b where
  total   = isoTotal
```

With the above technique, and using Haskell's instances, getting a `VerifiedOrd` instance for `Peano` reduces to definition of an `Iso Nat Peano`.

Proof Composition via Products Finally, we present a mechanism to automatically reduce proofs on product types to proofs of the product components. For example, lexicographic ordering preserves the ordering laws. First, we use class instances to define lexicographic ordering.

```
instance (VerifiedOrd a, VerifiedOrd b) ⇒ Ord (a, b) where
  (x1, y1) ≤ (x2, y2) = if x1 == x2 then y1 ≤ y2 else x1 ≤ x2
```

Then, we prove that lexicographic ordering preserves the ordering laws. For example, it preserves totality.

```
prodTotal :: (VerifiedOrd a, VerifiedOrd b)
          ⇒ p:(a, b) → q:(a, b) → {p ≤ q || q ≤ p}
prodTotal p@(x1, y1) q@(x2, y2)
 =  p ≤ q || q ≤ p
 =. if x1 == x2 then (y1 ≤ y2 || y2 ≤ y1) else True
    ∵ total x1 x2
 =. if x1 == x2 then True                  else True
    ∵ total y1 y2
 ** QED
```

Finally, using the `prodTotal` proof method, we conclude that each instance defined via the lexicographic ordering is indeed a verified instance.

```
instance (VerifiedOrd a, VerifiedOrd b) ⇒ VerifiedOrd (a, b) where
  total   = prodTotal
```

For example the type `(Peano, Peano)` is derived to be a `VerifiedOrd` instance.

In short, we can decompose an algebraic datatype into an isomorphic type using sums and products to generate verified instances for arbitrary Haskell datatypes. This could be combined with the Glasgow Haskell Compiler's (GHC) support for generics [61] to automate the derivation of verified instances for user datatypes. In §5.5, we use these ideas to develop fully safe interfaces to LVish modules, as well as verifying programming patterns from DPJ.

5.2 Refinement Reflection

Next, we formalize refinement reflection via a core calculus λ^R. We define a decidable SMT language λ^S to approximate the higher order, potentially diverging target language λ^R and present a decidable and sound type system for λ^R.

$$
\begin{array}{rrcl}
\textbf{\textit{Operators}} & \odot & ::= & = \mid < \\
\textbf{\textit{Constants}} & c & ::= & \wedge \mid \neg \mid \odot \mid +, -, \ldots \mid \texttt{True} \mid \texttt{False} \mid 0, -1, 1, \ldots \\
\textbf{\textit{Values}} & w & ::= & c \mid \lambda x.e \mid D\,\overline{w} \\
\textbf{\textit{Expressions}} & e & ::= & w \mid x \mid e\ e \mid \texttt{case}\ x = e\ \texttt{of}\ \{D\,\overline{x} \rightarrow e\} \\
\textbf{\textit{Binders}} & b & ::= & e \mid \texttt{let rec}\ x{:}\tau = b\ \texttt{in}\ b \\
\textbf{\textit{Program}} & p & ::= & b \mid \texttt{reflect}\ x{:}\tau = e\ \texttt{in}\ p \\
\textbf{\textit{Basic Types}} & B & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \\
\textbf{\textit{Refined Types}} & \tau & ::= & \{v{:}B^{[\Downarrow]} \mid r\} \mid x{:}\tau \rightarrow \tau
\end{array}
$$

Figure 5.1. Syntax of λ^R.

5.2.1 Syntax

Figure 5.1 summarizes the syntax of λ^R, which is essentially the calculus λ^U (from § 2.2) with explicit recursion and a special `reflect` binding to denote terms that are reflected into the refinement logic. The elements of λ^R are layered into primitive constants, values, expressions, binders and programs.

Constants The primitive constants of λ^R include all the primitive logical operators $\odot$, here, the set $\{=, <\}$. Moreover, they include the primitive booleans `True`, `False`, integers $-1, 0$, `1`, *etc.*, and logical operators $\wedge$, $\vee$, $\neg$, *etc.*.

Data Constructors Data constructors are special constants. For example, the data type [`Int`], which represents finite lists of integers, has two data constructors: [] (nil) and : (cons).

Values & Expressions The values of λ^R include constants, λ-abstractions $\lambda x.e$, and fully applied data constructors D that wrap values. The expressions of λ^R include values, variables x, applications $e\ e$, and `case` expressions.

Binders & Programs A *binder* b is a series of possibly recursive `let` definitions, followed by an expression. A *program* p is a series of `reflect` definitions, each of which names a function that is reflected into the refinement logic, followed by a binder. The stratification of programs via binders is required so that arbitrary recursive definitions are allowed in the program but cannot be inserted into the logic via refinements or reflection. (We *can* allow non-recursive `let` binders in expressions e, but omit them for simplicity).

5.2.2 Operational Semantics

We define $\hookrightarrow$ to be the small step, call-by-name β-reduction semantics for λ^R. We evaluate reflected terms as recursive `let` bindings, with extra termination-check constraints imposed by the type system:

$$\texttt{reflect}\ x{:}\tau = e\ \texttt{in}\ p \hookrightarrow \texttt{let rec}\ x{:}\tau = e\ \texttt{in}\ p$$

We define $\hookrightarrow^\star$ to be the reflexive, transitive closure of $\hookrightarrow^*$. Moreover, we define $\approx_\beta$ to be the reflexive, symmetric, and transitive closure of $\hookrightarrow^*$.

Constants Application of a constant requires the argument be reduced to a value; in a single step, the expression is reduced to the output of the primitive constant operation, *i.e.* $c\ v \hookrightarrow \delta(c,v)$. For example, consider $=$, the primitive equality operator on integers. We have $\delta(=,n)\ \doteq =_n$ where $\delta(=_n,m)$ equals `True` iff m is the same as n.

Equality We assume that the equality operator is defined *for all* values, and, for functions, is defined as extensional equality. That is, for all f and f', $(f = f') \hookrightarrow^*$ `True` iff $\forall v.\ f\ v \approx_\beta f'\ v$. We assume source *terms* only contain implementable equalities over non-function types; while function extensional equality only appears in *refinements* and is approximated by the underlying logic.

5.2.3 Types

λ^R types include basic types, which are *refined* with predicates, and dependent function types. *Basic types* B comprise integers, booleans, and a family of data-types T (representing lists, trees *etc.*). For example, the data type $[\texttt{Int}]$ represents lists of integers. We refine basic types with predicates (boolean-valued expressions e) to obtain *basic refinement types* $\{v{:}B \mid e\}$. We use $\Downarrow$ to mark provably terminating computations and use refinements to ensure that if $e{:}\{v{:}B^{\Downarrow} \mid e'\}$, then e terminates (as in chapter 2). Finally, we have dependent *function types* $x{:}\tau_x \to \tau$ where the input x has the type τ_x and the output τ may refer to the input binder x. We write B to abbreviate $\{v{:}B \mid \texttt{True}\}$, and $\tau_x \to \tau$ to abbreviate $x{:}\tau_x \to \tau$ if x does not appear in τ.

Constants For each constant c we define its type $\mathsf{Ty}(c)$, *e.g.*

$$\begin{aligned}
\mathsf{Ty}(3) &\doteq \{v\!:\!\texttt{Int} \mid v = 3\} \\
\mathsf{Ty}(+) &\doteq \texttt{x}\!:\!\texttt{Int} \rightarrow \texttt{y}\!:\!\texttt{Int} \rightarrow \{v\!:\!\texttt{Int} \mid v = x + y\} \\
\mathsf{Ty}(\leq) &\doteq \texttt{x}\!:\!\texttt{Int} \rightarrow \texttt{y}\!:\!\texttt{Int} \rightarrow \{v\!:\!\texttt{Bool} \mid v \Leftrightarrow x \leq y\}
\end{aligned}$$

5.2.4 Refinement Reflection

The key idea in our work is to *strengthen* the output type of functions with a refinement that *reflects* the definition of the function in the logic. We do this by treating each `reflect`-binder (`reflect` $f\!:\!\tau = e$ `in` p) as a `let rec`-binder with a reflected singleton type (`let rec` $f\!:\!\mathsf{Reflect}(\tau, e) = e$ `in` p) during type checking (rule T-REFLECT in Figure 5.3).

Reflection We write $\mathsf{Reflect}(\tau, e)$ for the *reflection* of the term e into the type τ, defined by strengthening τ as:

$$\begin{aligned}
\mathsf{Reflect}(\{v\!:\!B \mid r\}, e) &\doteq \{v\!:\!B \mid r \wedge v = e\} \\
\mathsf{Reflect}(x\!:\!\tau_x \rightarrow \tau, \lambda y.e) &\doteq x\!:\!\tau_x \rightarrow \mathsf{Reflect}(\tau, e[x/y])
\end{aligned}$$

As an example, recall from § 5.1 that the `reflect fib` strengthens the type of `fib` with the refinement `fibP`.

Consequences for Verification Reflection has two consequences for verification. First, the reflected refinement is *not trusted*; it is itself verified (as a valid output type) during type checking. Second, instead of being tethered to quantifier instantiation heuristics or having to program "triggers" as in Dafny [54] or F* [88], the programmer can predictably "unfold" the definition of the function during a proof simply by "calling" the function, which, as discussed in § 5.4, we have found to be a very natural way of structuring proofs.

5.2.5 The SMT logic λ^S

λ^R is a higher order, potentially diverging language that cannot be used for decidable verification. Next, we describe λ^S, a conservative, first order approximation of λ^R where higher order features are approximated with uninterpreted functions, yielding an SMT-based algorithmic

$$
\begin{array}{rrcl}
\textbf{\textit{Predicates}} & r & ::= & r \oplus_2 r \mid \oplus_1 r \mid n \mid b \mid x \mid D \mid x\,\overline{r} \\
 & & \mid & \texttt{if}\ r\ \texttt{then}\ r\ \texttt{else}\ r \\
\textbf{\textit{Integers}} & n & ::= & 0, -1, 1, \ldots \\
\textbf{\textit{Booleans}} & b & ::= & \texttt{True} \mid \texttt{False} \\
\textbf{\textit{Binary Operators}} & \oplus_2 & ::= & = \mid < \mid \wedge \mid + \mid - \mid \ldots \\
\textbf{\textit{Unary Operators}} & \oplus_1 & ::= & \neg \mid \ldots \\
\textbf{\textit{Sort Arguments}} & s_a & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{U} \mid \texttt{Fun}\ s_a\ s_a \\
\textbf{\textit{Sorts}} & s & ::= & s_a \to s
\end{array}
$$

Figure 5.2. Syntax of λ^S.

logic that enjoys soundness and decidability.

Syntax Figure 5.2 summarizes the syntax of λ^S, the *sorted* (SMT-) decidable logic of quantifier-free equality, uninterpreted functions and linear arithmetic (QF-EUFLIA) [69, 4]. The *terms* of λ^S include integers n, booleans b, variables x, data constructors D (encoded as constants), fully applied unary $\oplus_1$ and binary $\oplus_2$ operators, and application $x\,\overline{r}$ of an uninterpreted function x. The *sorts* of λ^S include the built-in `Int` and `Bool`. The interpreted functions of λ^S, *e.g.* the logical constants $=$ and $<$, have the function sort $s \to s$. Other functional values in λ^R, *e.g.* reflected λ^R functions and λ-expressions, have the first-order uninterpreted sort `Fun` s s. The universal sort `U` represents all other values.

5.2.6 Transforming λ^R into λ^S

A *type environment* Γ is a sequence of type bindings $x_1 : \tau_1, \ldots, x_n : \tau_n$. We use the type environment to define the judgment $\Gamma \vdash e \rightsquigarrow r$ that transforms a λ^R term e into a λ^S term r. Most of the transformation rules are identity and can be found in [96]. Here we discuss the non-identity ones.

Embedding Types We embed λ^R types into λ^S sorts as:

$$
\begin{array}{rclrcl}
(\!|\texttt{Int}|\!) & \doteq & \texttt{Int} & (\!|T|\!) & \doteq & \texttt{U} \\
(\!|\texttt{Bool}|\!) & \doteq & \texttt{Bool} & (\!|x : \tau_x \to \tau|\!) & \doteq & \texttt{Fun}\ (\!|\tau_x|\!)\ (\!|\tau|\!)
\end{array}
$$

Embedding Constants Elements shared on both λ^R and λ^S translate to themselves. These elements include booleans, integers, variables, binary and unary operators. SMT solvers do not support currying, and so in λ^S, all function symbols must be fully applied. Thus, we assume that all applications to primitive constants and data constructors are fully applied, *e.g.* by converting source terms like `(+ 1)` to `(\z → z+1)`.

Embedding Functions As λ^S is a first-order logic, we embed λ-abstraction using the uninterpreted function `lam`.

$$\frac{\Gamma, x : \tau_x \vdash e \rightsquigarrow r \qquad \Gamma \vdash (\lambda x.e) : (x : \tau_x \to \tau)}{\Gamma \vdash \lambda x.e \rightsquigarrow \mathtt{lam}^{(|\tau_x|)}_{(|\tau|)}\ x\ r} \text{ T-FUN}$$

The term $\lambda x.e$ of type $\tau_x \to \tau$ is transformed to $\mathtt{lam}^{s_x}_{s}\ x\ r$ of sort $\mathtt{Fun}\ s_x\ s$, where s_x and s are respectively $(|\tau_x|)$ and $(|\tau|)$, $\mathtt{lam}^{s_x}_{s}$ is a special uninterpreted function of sort $s_x \to s \to \mathtt{Fun}\ s_x\ s$, and x of sort s_x and r of sort s are the embedding of the binder and body, respectively. As `lam` is an SMT-function, it *does not* create a binding for x. Instead, x is renamed to a *fresh* name pre-declared in the SMT logic.

Embedding Applications Dually, we embed applications via defunctionalization [77] with the uninterpreted `app` function.

$$\frac{\Gamma \vdash e' \rightsquigarrow r' \qquad \Gamma \vdash e \rightsquigarrow r \qquad \Gamma \vdash e : \tau_x \to \tau}{\Gamma \vdash e\ e' \rightsquigarrow \mathtt{app}^{(|\tau_x|)}_{(|\tau|)}\ r\ r'} \text{ T-APP}$$

The term $e\ e'$, where e and e' have types $\tau_x \to \tau$ and τ_x, is transformed to $\mathtt{app}^{s_x}_{s}\ r\ r' : s$ where s and s_x are $(|\tau|)$ and $(|\tau_x|)$, the $\mathtt{app}^{s_x}_{s}$ is a special uninterpreted function of sort $\mathtt{Fun}\ s_x\ s \to s_x \to s$, and r and r' are the respective translations of e and e'.

Embedding Data Types We translate each data constructor to a predefined λ^S constant $\mathtt{s}_D$ of sort $(|\mathsf{Ty}(D)|)$.

$$\Gamma \vdash D \rightsquigarrow \mathtt{s}_D$$

For each datatype, we assume the existence of reflected functions that *check* the top-level constructor and *select* their individual fields. For example, for lists, we assume the existence of

measures:

```
isNil []     = True     isCons (x:xs) = True
isNil (x:xs) = False    isCons []     = False

sel1 (x:xs)  = x        sel2 (x:xs)   = xs
```

Due to the simplicity of their syntax the above checkers and selectors can be automatically instantiated in the logic (*i.e.* without actual calls to the reflected functions at source level) using the measure mechanism (§ 2.1.6).

To generalize, let D_i be a data constructor such that

$$\mathsf{Ty}(D_i) \doteq \tau_{i,1} \to \cdots \to \tau_{i,n} \to \tau$$

Then the *check function* $\texttt{is}_{D_i}$ has the sort $\texttt{Fun}\ (|\tau|)\ \texttt{Bool}$, and the *select function* $\texttt{sel}_{D_{i,j}}$ has the sort $\texttt{Fun}\ (|\tau|)\ (|\tau_{i,j}|)$.

We translate case-expressions of λ^R into nested `if` terms in λ^S, by using the check functions in the guards, and the select functions for the binders of each case.

$$\frac{\Gamma \vdash e \rightsquigarrow r \qquad \Gamma \vdash e_i[\overline{\texttt{sel}_{D_i}\ x}/\overline{y_i}][e/x] \rightsquigarrow r_i}{\Gamma \vdash \texttt{case}\ x = e\ \texttt{of}\ \{D_i\ \overline{y_i} \to e_i\} \rightsquigarrow \texttt{if app is}_{D_1}\ r\ \texttt{then}\ r_1\ \texttt{else}\ \ldots\ \texttt{else}\ r_n}\ \text{T-CASE}$$

For example, the body of the list append function

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

is reflected into the λ^S refinement:

$$\texttt{if isNil}\ xs\ \texttt{then}\ ys\ \texttt{else sel1}\ xs : (\texttt{sel2}\ xs\ +\!\!+\ ys)$$

We favor selectors to the axiomatic translation of HALO [103] to avoid universally quantified formulas and the resulting instantiation unpredictability.

Typing $\boxed{\Gamma \vdash p : \tau}$

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{T-Var} \qquad \frac{}{\Gamma \vdash c : \mathsf{Ty}(c)}\ \textsc{T-Con}$$

$$\frac{\Gamma \vdash p : \tau' \quad \Gamma \vdash \tau' \preceq \tau}{\Gamma \vdash p : \tau}\ \textsc{T-Sub} \qquad \frac{\Gamma \vdash e : \{v{:}B \mid r_r\}}{\Gamma \vdash e : \{v{:}B \mid r_r \wedge v = e\}}\ \textsc{T-Exact}$$

$$\frac{\Gamma, x{:}\tau_x \vdash e : \tau}{\Gamma \vdash \lambda x.e : x{:}\tau_x \to \tau}\ \textsc{T-Fun} \qquad \frac{\Gamma \vdash e_1 : (x{:}\tau_x \to \tau) \quad \Gamma \vdash e_2 : \tau_x}{\Gamma \vdash e_1\ e_2 : \tau}\ \textsc{T-App}$$

$$\frac{\Gamma, x{:}\tau_x \vdash b_x : \tau_x \quad \Gamma, x{:}\tau_x \vdash \tau_x \quad \Gamma, x{:}\tau_x \vdash b : \tau \quad \Gamma \vdash \tau}{\Gamma \vdash \texttt{let rec}\ x{:}\tau_x = b_x\ \texttt{in}\ b : \tau}\ \textsc{T-Let}$$

$$\frac{\Gamma \vdash \texttt{let rec}\ f{:}\mathsf{Reflect}(\tau_f, e) = e\ \texttt{in}\ p : \tau}{\Gamma \vdash \texttt{reflect}\ f{:}\tau_f = e\ \texttt{in}\ p : \tau}\ \textsc{T-Reflect}$$

$$\frac{\begin{array}{cc}\Gamma \vdash e : \{v{:}T \mid e_r\} & \Gamma \vdash \tau \\ \forall i.\mathsf{Ty}(D_i) = \overline{y_j{:}\tau_j} \to \{v{:}T \mid e_{r_i}\} & \Gamma, \overline{y_j{:}\tau_j}, x{:}\{v{:}T \mid e_r \wedge e_{r_i}\} \vdash e_i : \tau\end{array}}{\Gamma \vdash \texttt{case}\ x = e\ \texttt{of}\ \{D_i\ \overline{y_i} \to e_i\} : \tau}\ \textsc{T-Case}$$

Well Formedness $\boxed{\Gamma \vdash \tau}$

$$\frac{\Gamma, v{:}B \vdash e : \mathtt{Bool}^{\Downarrow}}{\Gamma \vdash \{v{:}B \mid e\}}\ \textsc{WF-Base} \qquad \frac{\Gamma \vdash \tau_x \quad \Gamma, x{:}\tau_x \vdash \tau}{\Gamma \vdash x{:}\tau_x \to \tau}\ \textsc{WF-Fun}$$

Subtyping $\boxed{\Gamma \vdash \tau_1 \preceq \tau_2}$

$$\frac{\Gamma' \doteq \Gamma, v{:}\{v{:}B^{\Downarrow} \mid e\} \quad \Gamma' \vdash e' \rightsquigarrow r' \quad \mathsf{SmtValid}((|\Gamma'|) \Rightarrow r')}{\Gamma \vdash \{v{:}B \mid e\} \preceq \{v{:}B \mid e'\}}\ \preceq\textsc{-Base}$$

$$\frac{\Gamma \vdash \tau'_x \preceq \tau_x \quad \Gamma, x{:}\tau'_x \vdash \tau \preceq \tau'}{\Gamma \vdash x{:}\tau_x \to \tau \preceq x{:}\tau'_x \to \tau'}\ \preceq\textsc{-Fun}$$

Figure 5.3. Type checking of λ^R.

5.2.7 Typing Rules

Next, we present the typing, well-formedness, and subtyping [51, 100] rules of λ^R.

Typing A judgment $\Gamma \vdash p : \tau$ states that the program p has the type τ in the environment Γ. That is, when the free variables in p are bound to expressions described by Γ, the program p will evaluate to a value described by τ.

Rules All but two of the rules are standard [51, 100]. First, rule T-REFLECT is used to strengthen

the type of each reflected binder with its definition, as described previously in § 5.2.4. Second, rule T-EXACT strengthens the expression with a singleton type equating the value and the expression (*i.e.* reflecting the expression in the type). This is a generalization of the "selfification" rules from [72, 51], and is required to equate the reflected functions with their definitions. For example, the application $(\texttt{fib}\ \ 1)$ is typed as $\{v\!:\!\texttt{Int} \mid \texttt{fibP}\ v\ 1 \wedge v = \texttt{fib}\ \ 1\}$ where the first conjunct comes from the (reflection-strengthened) output refinement of `fib` § 5.1 and the second comes from rule T-EXACT.

Well-formedness A judgment $\Gamma \vdash \tau$ states that the refinement type τ is well-formed in the environment Γ. Following chapter 2, the type τ is well-formed if all the refinements in τ are `Bool`-typed, provably terminating expressions in Γ.

Subtyping A judgment $\Gamma \vdash \tau_1 \preceq \tau_2$ states that the type τ_1 is a subtype of τ_2 in the environment Γ. Informally, τ_1 is a subtype of τ_2 if the refinement of τ_1 *implies* the refinement of τ_2 under the assumptions described by Γ. Subtyping of basic types reduces to implication checking.

Verification Conditions The implication or *verification condition* (VC) $(|\Gamma|) \Rightarrow r$ is *valid* only if the set of values described by Γ, is subsumed by the set of values described by r. Γ is embedded into logic by conjoining (the embeddings of) the refinements of provably terminating binders (Chapter 2):

$$(|\Gamma|) \doteq \bigwedge_{x \in \Gamma} (|\Gamma, x|)$$

where we embed each binder as

$$(|\Gamma, x|) \doteq \begin{cases} r & \text{if } \Gamma(x) = \{v\!:\!B^{\Downarrow} \mid e\},\ \Gamma \vdash e[x/v] \rightsquigarrow r \\ \texttt{True} & \text{otherwise.} \end{cases}$$

It is important to note that since λ^S is carefully restricted to SMT-decidable theories, VC checking, and thus type checking of λ^R, is decidable.

5.2.8 Soundness

Following λ^U from chapter 2, in [96], we show that if validity checking respects the axioms of β-equivalence, then λ^R is sound.

We define the β-equivalence axioms on the uninterpreted function that represent λ-abstraction (`lam`) and and application (`app`).

$$\begin{aligned} \forall x\ y\ e.\texttt{lam}\ x\ e &= \texttt{lam}\ y\ (e[y/x]) \\ \forall x\ e_x\ e.(\texttt{app}\ (\texttt{lam}\ x\ e)\ e_x) &= e[e_x/x] \end{aligned}$$

We prove that when validity checking assumes the β-equivalence axioms, λ^R is sound.

Theorem 6. *[Soundness of λ^R] Assuming the β-equivalence axioms, if $\emptyset \vdash p : \tau$ and $p \hookrightarrow^\star w$ then $\emptyset \vdash w : \tau$.*

Theorem 6 lets us interpret well typed terminating programs as proofs of propositions. For example, in § 5.1 we verified that $\texttt{fibUp} :: n{:}\texttt{Nat} \rightarrow \{\texttt{fib}\ \texttt{n} \leq \texttt{fib}\ (\texttt{n}+1)\}$. Via soundness of λ^R, we get runtime monotonicity of `fib`.

$$\forall n.0 \leq n \hookrightarrow^\star \texttt{True} \Rightarrow \texttt{fib}\ n \leq \texttt{fib}\ (n+1) \hookrightarrow^\star \texttt{True}$$

Approximation of β-equivalence Though sound and precise, directly extending the logic with β-equivalence axioms would render SMT validity checking undecidable. Next, we discuss an incomplete, yet decidable, technique that allows the user to manually instantiate the β-equivalence axioms when required for precise typing.

5.3 Reasoning About Lambdas

Soundness and precision of λ^R relies on the β-equivalence and extensionality axioms that are undecidable. Next, we present a decidable but incomplete way to approximate β-equivalence by strengthening the VCs with equalities § 5.3.1, and extensionality by introducing a combinator for safely asserting extensional equality § 5.3.2. In the rest of this section, for clarity we omit `app`

when it is clear from the context.

5.3.1 Equivalence

As soundness relies on axioms of β-equivalence we can safely *instantiate* the axioms of α- and β-equivalence on any set of terms of our choosing and still preserve soundness. That is, instead of checking the validity of a VC $p \Rightarrow q$, we check the validity of a *strengthened VC*, $a \Rightarrow p \Rightarrow q$, where a is a (finite) conjunction of *equivalence instances* derived from p and q, as discussed below.

Representation Invariant The lambda binders, for each SMT sort, are drawn from a pool of names x_i where the index $i = 1, 2, \ldots$. When representing λ terms we enforce a *normalization invariant* that for each lambda term $\texttt{lam}\ x_i\ e$, the index i is greater than any lambda argument appearing in e.

α-instances For each syntactic term $\texttt{lam}\ x_i\ e$, and λ-binder x_j such that $i < j$ appearing in the VC, we generate an *α-equivalence instance predicate* (or *α-instance*):

$$\texttt{lam}\ x_i\ e = \texttt{lam}\ x_j\ e[x_j/x_i]$$

The conjunction of α-instances can be more precise than De Bruijn representation, as they let the SMT solver deduce more equalities via congruence. For example, consider the VC needed to prove the applicative laws for `Reader`:

$$d = \texttt{lam}\ x_1\ (x\ x_1) \Rightarrow \texttt{lam}\ x_2\ ((\texttt{lam}\ x_1\ (x\ x_1))\ x_2) = \texttt{lam}\ x_1\ (d\ x_1)$$

The α instance $\texttt{lam}\ x_1\ (d\ x_1) = \texttt{lam}\ x_2\ (d\ x_2)$ derived from the VC's hypothesis, combined with congruence immediately yields the VC's consequence.

β-instances For each syntactic term $\texttt{app}\ (\texttt{lam}\ x\ e)\ e_x$, with e_x not containing any λ-abstractions, appearing in the VC, we generate an *β-equivalence instance predicate* (or *β-instance*):

$$\texttt{app}\ (\texttt{lam}\ x_i\ e)\ e_x = e[e_x/x_i],\ \text{s.t. } e_x \text{ is } \lambda\text{-free}$$

We require the λ-free restriction as a simple way to enforce that the reduced term $e[e'/x_i]$ enjoys the representation invariant.

For example, consider the following VC needed to prove that the bind operator for lists satisfies the monadic associativity law.

$$(f\ x \gg\!= g) = \texttt{app}\ (\texttt{lam}\ y\ (f\ y \gg\!= g))\ x$$

The right-hand side of the above VC generates a β-instance that corresponds directly to the equality, allowing the SMT solver to prove the (strengthened) VC.

Normalization The combination of α- and β-instances is often required to discharge proof obligations. For example, when proving that the bind operator for the `Reader` monad is associative, we need to prove the VC:

$$\texttt{lam}\ x_2\ (\texttt{lam}\ x_1\ w) = \texttt{lam}\ x_3\ (\texttt{app}\ (\texttt{lam}\ x_2\ (\texttt{lam}\ x_1\ w))\ w)$$

The SMT solver proves the VC via the equalities corresponding to an α and then β-instance:

$$\begin{aligned}\texttt{lam}\ x_2\ (\texttt{lam}\ x_1\ w) &=_\alpha \texttt{lam}\ x_3\ (\texttt{lam}\ x_1\ w)\\ &=_\beta \texttt{lam}\ x_3\ (\texttt{app}\ (\texttt{lam}\ x_2\ (\texttt{lam}\ x_1\ w))\ w)\end{aligned}$$

5.3.2 Extensionality

Often, we need to prove that two functions are equal, given the definitions of reflected binders. For example, consider

```
reflect id
id x = x
```

LIQUID HASKELL accepts the proof that `id x = x` for all `x`:

```
id_x_eq_x :: x:a → {id x = x}
id_x_eq_x = \x → id x =. x ** QED
```

as "calling" `id` unfolds its definition, completing the proof. However, consider this η-expanded variant of the above proposition:

```
type Id_eq_id = {(\x → id x) = (\y → y)}
```

LIQUID HASKELL *rejects* the proof:

```
fails :: Id_eq_id
fails =  (\x → id x) =. (\y → y) ** QED
```

The invocation of `id` unfolds the definition, but the resulting equality refinement `{id x = x}` is *trapped* under the λ-abstraction. That is, the equality is absent from the typing environment at the *top* level, where the left-hand side term is compared to `\y → y`. Note that the above equality requires the definition of `id` and hence is outside the scope of purely the α- and β-instances.

An Exensionality Operator To allow function equality via extensionality, we provide the user with a (family of) *function comparison operator(s)* that transform an *explanation* `p` which is a proof that `f x = g x` for every argument `x`, into a proof that `f = g`.

```
=∀ :: f:(a → b) → g:(a → b)
   → exp:(x:a → {f x = g x})
   → {f = g}
```

Of course, =∀ cannot be implemented; its type is *assumed*. We can use =∀ to prove `Id_eq_id` by providing a suitable explanation:

```
pf_id_id :: Id_eq_id
pf_id_id = (\y → y) =∀ (\x → id x) ∴ expl ** QED
 where
   expl  = (\x → id x =. x ** QED)
```

The explanation is the second argument to $\therefore$ which has the following type that syntactically fires β-instances:

```
x:a → {(\x → id x) x = ((\x → x) x}
```

Table 5.1. Summary of Refinement Reflection Case Studies.

CATEGORY			LOC
I.	**Arithmetic**		
	Fibonacci	§ 5.1	48
	Ackermann	[92] , Fig. 5.4	280
II.	**Algebraic Data Types**		
	Fold Universal	[67]	105
	Fold Fusion	[67]	
III.	**Typeclasses**	Table 5.2	
	Monoid	Peano, Maybe, List	189
	Functor	Maybe, List, Id, Reader	296
	Applicative	Maybe, List, Id, Reader	578
	Monad	Maybe, List, Id, Reader	435
IV.	**Functional Correctness**		
	SAT Solver	[15]	133
	Unification	[85]	200
V.	**Deterministic Parallelism**		
	Concurrent Sets	§ 5.5.1	906
	n-body simulation	§ 5.5.2	930
	Parallel Reducers	§ 5.5.3	55
TOTAL			4155

5.4 Evaluation

We have implemented refinement reflection in LIQUID HASKELL. In this section, we evaluate our approach by using LIQUID HASKELL to verify a variety of deep specifications of Haskell functions drawn from the literature and categorized in Table 5.1, totalling about 4000 lines of specifications and proofs. Next, we detail each of the first four classes of specifications, illustrate how they were verified using refinement reflection, and discuss the strengths and weaknesses of our approach. *All* of these proofs require refinement reflection, *i.e.* are beyond the scope of shallow refinement typing.

Proof Strategies. Our proofs use three building blocks, that are seamlessly connected via refine-

ment typing:

- ***Un/folding*** definitions of a function `f` at arguments `e1...en`, which due to refinement reflection, happens whenever the term `f e1 ... en` appears in a proof. For exposition, we render the function whose un/folding is relevant as `f`;
- ***Lemma Application*** which is carried out by using the "because" combinator $(\cdot.\cdot)$ to instantiate some fact at some inputs;
- ***SMT Reasoning*** in particular, *arithmetic*, *ordering* and *congruence closure* which kicks in automatically (and predictably!), allowing us to simplify proofs by not having to specify, *e.g.* which subterms to rewrite.

5.4.1 Arithmetic Properties

The first category of theorems pertains to the textbook Fibonacci and Ackermann functions. The former were shown in § 5.1. The latter are summarized in Figure 5.4, which shows two alternative definitions for the Ackermann function. We proved equivalence of the definition (Prop 1) and various arithmetic relations between them (Prop 2 — 13), by mechanizing the proofs from [92].

Monotonicity Prop 3. shows that $A_n(x)$ is increasing on x. We derived Prop 4. by applying `fMono` theorem from § 5.1 with input function the partially applied Ackermann Function $A_n(\star)$. Similarly, we derived the monotonicity Prop 9. by applying `fMono` to the locally increasing Prop. 8 and $A^h_n(\star)$. Prop 5. proves that $A_n(x)$ is increasing on the *first* argument n. As `fMono` applies to the *last* argument of a function, we cannot directly use it to derive Prop 6. Instead, we define a variant `fMono2` that works on the first argument of a binary function, and use it to derive Prop 6.

Constructive Proofs In [92] Prop 12. was proved by constructing an auxiliary *ladder* that counts the number of (recursive) invocations of the Ackermann function, and uses this count to bound $A^h_n(x)$ and $A_n(x)$. It turned out to be straightforward and natural to formalize the proof just by defining the `ladder` function in Haskell, reflecting it, and using it to formalize the algebra from [92].

Ackermann's Function

$$A_n(x) \doteq \begin{cases} x+2 & \text{, if } n=0 \\ 2 & \text{, if } x=0 \\ A_{n-1}(A_n(x-1)) \end{cases} \qquad A_n^h(x) \doteq \begin{cases} x & \text{, if } h=0 \\ A_n(A_n^{h-1}(x)) \end{cases}$$

Properties

$$\begin{array}{lrcl}
1. & & A_{n+1}(x) & = & A_n^x(2) \\
2. & & x+1 & < & A_n(x) \\
3. & & A_n(x) & < & A_n(x+1) \\
4. & x<y & A_n(x) & < & A_n(y) \\
5. & 0<x & A_n(x) & < & A_{n+1}(x) \\
6. & 0<x, n<m & A_n(x) & < & A_m(x) \\
7. & & A_n^h(x) & < & A_n^{h+1}(x) \\
8. & & A_n^h(x) & < & A_n^h(x+1) \\
9. & x<y & A_n^h(x) & < & A_n^h(y) \\
10. & & A_n^h(x) & < & A_{n+1}^h(x) \\
11. & 0<n, l-2<x & x+l & < & A_n(x) \\
12. & 0<n, l-2<x & A_n^l(x) & < & A_{n+1}(x) \\
13. & & A_n^x(y) & < & A_{n+1}(x+y)
\end{array}$$

Figure 5.4. Ackermann Properties [92], $\forall n, m, x, y, h, l \geq 0$

5.4.2 Algebraic Data Properties

The second category of properties pertain to algebraic data types.

Fold Univerality Next, we proved properties of list folding, such as the following, describing the *universal* property of right-folds [67]:

```
foldr_univ
  :: f:(a → b → b)
  → h:([a] → b)
  → e:b
  → ys:[a]
  → base:{h [] = e }
  → stp:(x:a →l:[a]→{h(x:l) = f x (h l)})
  → {h ys = foldr f e ys}
```

Our proof `foldr_univ` differs from the one in Agda, in two ways. First, we encode Agda's universal quantification over `x` and `l` in the assumption `stp` using a function type. Second, unlike

Agda, LIQUID HASKELL does not support implicit arguments, so at *uses* of `foldr_univ` the programmer must explicitly provide arguments for `base` and `stp`, as illustrated below.

Fold Fusion Let us define the usual composition operator:

```
reflect . :: (b → c) → (a → b) → a → c
f . g     = \x → f (g x)
```

We can prove the following `foldr_fusion` theorem (that shows operations can be pushed inside a `foldr`), by applying `foldr_univ` to explicit `bas` and `stp` proofs:

```
foldr_fusion
 :: h:(b → c)
 → f:(a → b → b)
 → g:(a → c → c)
 → e:b → z:[a] → x:a → y:b
 → fuse: {h (f x y) = g x (h y)})
 → {(h . foldr f e) z = foldr g (h e) z}

foldr_fusion h f g e ys fuse
  = foldr_univ g (h . foldr f e) (h e) ys
      (fuse_base h f e)
      (fuse_step h f e g fuse)
```

where `fuse_base` and `fuse_step` prove the base and inductive cases. For example the type of `fuse_base` is the following theorem

```
fuse_base :: h:(b→c) → f:(a→b→b) → e:b
          → {(h . foldr f e) [] = h e}
```

5.4.3 Typeclass Laws

We used LIQUID HASKELL to prove the Monoid, Functor, Applicative and Monad Laws, summarized in Table 5.2, for various user-defined instances summarized in Table 5.1.

Monoid Laws A Monoid is a datatype equipped with an associative binary operator $\diamondsuit$ and an *identity* element `mempty`. We use LIQUID HASKELL to prove that `Peano` (with `add` and `Z`), `Maybe` (with a suitable $\diamondsuit$ and `Nothing`), and `List` (with append `++` and `[]`) satisfy the monoid

Table 5.2. Typeclass Laws verified using LIQUID HASKELL.

	Monoid
Left Ident.	`mempty` $x \diamondsuit \equiv x$
Right Ident.	$x \diamondsuit$ `mempty` $\equiv x$
Associativity	$(x \diamondsuit y) \diamondsuit z \equiv x \diamondsuit (y \diamondsuit z)$
	Functor
Ident.	`fmap id` $xs \equiv$ `id` xs
Distribution	`fmap` $(g \circ h)$ $xs \equiv$ (`fmap` $g \circ$ `fmap` h) xs
	Applicative
Ident.	`pure id` $\circledast$ $v \equiv v$
Compos.	`pure` $(\circ) \circledast u \circledast v \circledast w \equiv u \circledast (v \circledast w)$
Homomorph.	`pure` $f \circledast$ `pure` $x \equiv$ `pure` $(f\ x)$
Interchange	$u \circledast$ `pure` $y \equiv$ `pure` $(\$\ y) \circledast u$
	Monad
Left Ident.	`return` $a \gg\!= f \equiv f\ a$
Right Ident.	$m \gg\!=$ `return` $\equiv m$
Associativity	$(m \gg\!= f) \gg\!= g \equiv m \gg\!= (\lambda x \rightarrow f\ x \gg\!= g)$

laws. For example, we prove that ++ (§ 5.4.3) is associative by reifying the textbook proof [42] into a Haskell function, where the induction corresponds to case-splitting and recurring on the first argument:

```
assoc :: xs:[a] → ys:[a] → zs:[a] → {(xs ++ ys) ++ zs = xs ++ (ys ++ zs)}


assoc [] ys zs     = ([] ++ ys) ++ zs
                   =. [] ++ (ys ++ zs)
                   ** QED
assoc (x:xs) ys zs = ((x:  xs)++ ys) ++ zs
                   =. (x: (xs ++ ys))++ zs
                   =.  x:((xs ++ ys) ++ zs)
                   =.  x: (xs ++ (ys ++ zs))
                       ∵ assoc xs ys zs
                   =. (x:xs)  ++ (ys ++ zs)
                   ** QED
```

Functor Laws A type is a functor if it has a function `fmap` that satisfies the *identity* and *distribution*

(or fusion) laws in Table 5.2. For example, consider the proof of the `fmap` distribution law for the lists, also known as "map-fusion", which is the basis for important optimizations in GHC [104]. We reflect the definition of `fmap`:

```
reflect map :: (a → b) → [a] → [b]
map f []     = []
map f (x:xs) = f x : fmap f xs
```

and then specify fusion and verify it by an inductive proof:

```
map_fusion :: f:(b → c) → g:(a → b) → xs:[a]
           → {map (f . g) xs = (map f . map g) xs}
```

Monad Laws The monad laws, which relate the properties of the two operators $\gg=$ and `return` (Table 5.2), refer to λ-functions, thus their proof exercises our support for defunctionalization and η- and β-equivalence. For example, consider the proof of the associativity law for the list monad. First, we reflect the bind operator:

```
reflect (>>=) :: [a] → (a → [b]) → [b]
(x:xs) >>= f = f x ++ (xs >>= f)
[]     >>= f = []
```

Next, we define an abbreviation for the associativity property:

```
type AssocLaw m f g = {m >>= f >>= g = m >>= (\x → f x >>= g)}
```

Finally, we can prove that the list-bind is associative:

```
assoc :: m:[a] → f:(a →[b]) → g:(b →[c]) → AssocLaw m f g
assoc [] f g
  =  [] >>= f >>= g
  =. [] >>= g
  =. []
  =. [] >>= (\x → f x >>= g) ** QED


assoc (x:xs) f g
  =  (x:xs) >>= f  >>= g
  =. (f x ++ xs >>= f) >>= g
  =. (f x >>= g) ++ (xs >>= f >>= g)
```

```
   ∵ bind_append (f x) (xs >>= f) g
=. (f x >>= g) ++ (xs >>= \y → f y >>= g)
   ∵ assoc xs f g
=. (\y → f y >>= g) x ++ (xs >>= \y → f y >>= g)
   ∵ βeq f g x
=. (x:xs) >>= (\y → f y >>= g) ** QED
```

Where the bind-append fusion lemma states that:

```
bind_append :: xs:[a] → ys:[a] → f:(a → [b])
            → {(xs++ys) >>= f = (xs >>= f)++(ys >>= f)}
```

Notice that the last step requires β-equivalence on anonymous functions, which we get by explicitly inserting the redex in the logic, via the following lemma with `trivial` proof

```
βeq :: f:_ → g:_ → x:_ → {bind (f x) g = (\y → bind (f y) g) x}
βeq _ _ _ = trivial
```

5.4.4 Functional Correctness

Finally, we proved correctness of two programs from the literature: a SAT solver and a Unification algorithm.

SAT Solver We implemented and verified the simple SAT solver used to illustrate and evaluate the features of the dependently typed language Zombie [15]. The solver takes as input a formula `f` and returns an assignment that *satisfies* `f` if one exists.

```
solve :: f:Formula → Maybe {a:Asgn|sat a f}
solve f = find (`sat` f) (assignments f)
```

Function `assignments f` returns all possible assignments of the formula `f` and `sat a f` returns `True` iff the assignment `a` satisfies the formula `f`:

```
reflect sat :: Asgn → Formula → Bool
assignments :: Formula → [Asgn]
```

Verification of `solve` follows simply by reflecting `sat` into the refinement logic, and using (bounded) refinements to show that `find` only returns values on which its input predicate yields `True` from chapter 4.

```
find :: p:(a → Bool) → [a] → Maybe {v:a | p v}
```

Unification As another example, we verified the unification of first order terms, as presented in [85]. First, we define a predicate alias for when two terms `s` and `t` are equal under a substitution `su`:

```
eq_sub su s t = apply su s == apply su t
```

Now, we can define a Haskell function `unify s t` that can diverge, or return `Nothing`, or return a substitution `su` that makes the terms equal:

```
unify :: s:Term → t:Term → Maybe {su| eq_sub su s t}
```

For the specification and verification we only needed to reflect `apply` and not `unify`; thus we only had to verify that the former terminates, and not the latter.

As before, we prove correctness by invoking separate helper lemmas. For example to prove the post-condition when unifying a variable `TVar i` with a term `t` in which `i` *does not* appear, we apply a lemma `not_in`:

```
unify (TVar i) t2
  | not (i Set_mem freeVars t2)
  = Just (const [(i, t2)] ∵ not_in i t2)
```

i.e. if `i` is not free in `t`, the singleton substitution yields `t`:

```
not_in :: i:Int
       → t:{Term | not (i Set_mem freeVars t)}
       → {eq_sub [(i, t)] (TVar i) t}
```

5.5 Verified Deterministic Parallelism

Finally, we evaluate our deterministic parallelism prototypes. Aside from the lines of proof code added, we evaluate the impact on runtime performance. Were we using a proof tool external to Haskell, this would not be necessary. But our proofs are Haskell programs—they are necessarily visible to the compiler. In particular, this means a proliferation of unit values and functions returning unit values. Also, typeclass instances are witnessed at runtime by "dictionary"

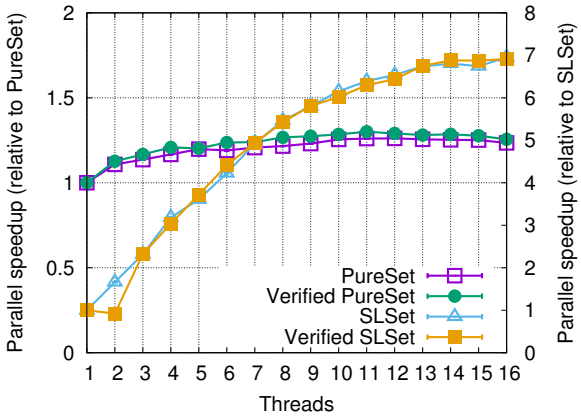

Figure 5.5. Parallel speedup for doing 1 million parallel inserts over 10 iterations, verified and unverified, relative to the unverified version, for PureSet and SLSet.

data structures passed between functions. Layering proof methods on top of existing classes like `Ord` (from § 5.1.4) could potentially add indirection or change the code generated, depending on the details of the optimizer. In our experiments we find little or no effect on runtime performance. Benchmarks were run on a single-socket Intel® Xeon® CPU E5-2699 v3 with 18 physical cores and 64GiB RAM.

5.5.1 LVish: Concurrent Sets

First, we use the `verifiedInsert` operation (from § 5.1.4) to observe the runtime slowdown imposed by the extra proof methods of `VerifiedOrd`. We benchmark concurrent sets storing 64-bit integers. Figure 5.5 compares the parallel speedups for a fixed number of parallel `insert` operations against parallel `verifiedInsert` operations, varying the number of concurrent threads. There is a slight observable difference between the two lines because the extra proof methods do exist at runtime. We repeat the experiment for two set implementations: a concurrent skiplist (SLSet) and a purely functional set inside an atomic reference (PureSet) as described in [52].

5.5.2 Monad-par: *n*-body simulation

Next, we verify deterministic behavior of an *n*-body simulation program that leverages `monad-par`, a Haskell library which provides deterministic parallelism for pure code [62].

Each simulated particle is represented by a type `Body` that stores its position, velocity and mass. The function `accel` computes the relative acceleration between two bodies:

```
accel :: Body → Body → Accel
```

where `Accel` represents the three-dimensional acceleration

```
data Accel = Accel Real Real Real
```

To compute the total acceleration of a body `b` we (1) compute the relative acceleration between `b` and each body of the system (`Vec Body`) and (2) we add each acceleration component. For efficiency, we use a parallel `mapReduce` for the above computation that first *maps* each vector body to get the acceleration relative to `b` (`accel b`) and then adds each `Accel` value by pointwise addition. `mapReduce` is only deterministic if the element is a `VerifiedMonoid` from § 5.1.4.

```
mapReduce :: VerifiedMonoid b 1⇒ (a1→b) 1→ Vec a 1→ b
```

To prove the determinism of an *n*-body simulation, we need to provide a `VerifiedMonoid` instance for `Accel`. We can easily prove that (`Real`, `+`, `0.0`) is a monoid. By product proof composition, we get a verified monoid instance for

```
type Accel′ = (Real, (Real, Real))
```

which is isomorphic to `Accel` (*i.e.* `Iso Accel′ Accel`).

Figure 5.6 shows the results of running two versions of the *n*-body simulation with 2,048 bodies over 5 iterations, with and without verification, using floating point doubles for `Real`[1]. Notably, the two programs have almost identical runtime performance. This demonstrates that even when verifying code that is run in a tight loop (like `accel`), we can expect that our programs will not be slowed down by an unacceptable amount.

[1] Floating point numbers notoriously violate associativity, but we use this approximation because Haskell does net yet have an implementation of *superaccumulators* [18].

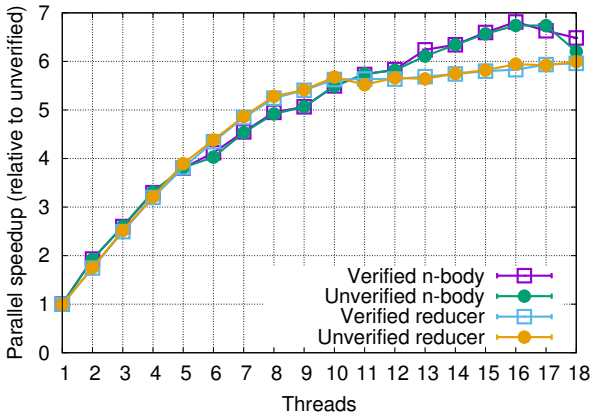

Figure 5.6. Parallel speedup for doing a parallel n-body simulation and parallel array reduction. The speedup is relative to the unverified version of each respective class of program.

5.5.3 DPJ: Parallel Reducers

The Deterministic Parallel Java (DPJ) project provides a deterministic-by-default semantics for the Java programming language [10]. In DPJ, one can declare a method as `commutative` and thus *assert* that racing instances of that method result in a deterministic outcome. For example:

```
commutative void updateSum(int n) writes R
  { sum += n; }
```

But, DPJ provides no means to formally prove commutativity and thus determinism of parallel reduction. In Liquid Haskell, we specified commutativity as an extra proof method that extends the `VerifiedMonoid` class.

```
class VerifiedMonoid a ⇒ VerifiedComMonoid a where
  commutes :: x:a → y:a → { x ♢ y = y ♢ x }
```

Provably commutative appends can be used to deterministically update a reducer variable, since the result is the same regardless of the order of appends. We used LVish [52] to encode a reducer variable with a value `a` and a region `s` as `RVar s a`.

```
newtype RVar s a
```

We specify that safe (*i.e.* deterministic) parallel updates require provably commutative appending.

```
updateRVar :: VerifiedComMonoid a ⇒ a → RVar s a → Par s ()
```

Following the DPJ program, we used `updateRVar`'s provably deterministic interface to compute, in parallel, the sum of an array with $3x10^9$ elements by updating a single, global reduction variable using a varying number of threads. Each thread sums segments of an array, sequentially, and updates the variable with these partial sums. In Figure 5.6, we compare the verified and unverified versions of our implementation to observe no appreciable difference in performance.

5.6 Conclusion

We presented *refinement reflection*, a method to extend *legacy* languages—with highly tuned libraries, compilers, and run-times—into theorem provers. The key idea is to reflect the code implementing a user-defined function into the function's (output) refinement type. As a consequence, at *uses* of the function, the function definition is unfolded into the refinement logic in a precise and predictable manner. We have implemented our approach in LIQUID HASKELL thereby retrofitting theorem proving into Haskell. We showed how to use reflection to verify that many widely used instances of the Monoid, Applicative, Functor and Monad typeclasses actually satisfy key algebraic laws needed to making the code using the typeclasses safe. Finally, transforming a mature language—with highly tuned parallel runtime—into a theorem prover enables us to build the first *deterministic parallelism library* that verifies assumptions about associativity and ordering—that are crucial for determinism but simply assumed by existing systems.

Acknowledgments The material of this chapter have been submitted for publication as it may appear in PLDI 2017: Vazou, Niki; Choudhury, Vikraman; Scott, Ryan G.; Newton, Ryan R.; Jhala, Ranjit. "Refinement Reflection: Parallel Legacy Languages as Theorem Provers".

Chapter 6

Case Study: Parallel String Matcher

> The way the processor industry is going, is to add more and more cores,
> but nobody knows how to program those things.
>
> – *Steve Jobs*

In this chapter, we prove correctness of parallelization of a naïve string matcher using Haskell as a theorem prover. We use refinement types to specify correctness properties, Haskell terms to express proofs – via Refinement Reflection from chapter 5– and LIQUID HASKELL to check correctness of proofs.

Optimization of sequential functions via parallelization is a well studied technique [43, 9]. Paper and pencil proofs have been developed to support the correctness of the transformation [17]. However, these paper written proofs show correctness of the parallelization algorithm and do not reason about the actual implementation that may end up being buggy.

Dependent Type Systems (like Coq [8] and Adga [71]) enable program equivalence proofs for the actual implementation of the functions to be parallelized. For example, SyDPaCC [60] is a Coq extension that given a naïve Coq implementation of a function, returns an Ocaml parallelized version with a proof of program equivalence. The limitation of this approach is that the initial function should be implemented in the specific dependent type framework and thus cannot use features and libraries from one's favorite programming language.

In chapter 5 we claimed that Refinement Reflection can turn any programming language into a proof assistant. In this chapter we check our claim and use LIQUID HASKELL to prove

program equivalence. Specifically, we *define in Haskell* a sequential string matching function, `toSM`, and its parallelization, `toSMPar`, using existing Haskell libraries; then, we *prove in Haskell* that these two functions are equivalent; finally, we check our proofs using LIQUID HASKELL.

Theorems as Refinement Types Refinement Types refine types with properties drawn from decidable logics. For example, the type `{v:Int | 0 < v}` describes all integer values `v` that are greater than `0`. We refine the unit type to express theorems, define unit value terms to express proofs, and use LIQUID HASKELL to check that the proofs prove the theorems. For example, LIQUID HASKELL accepts the type assignment `() :: {v:()| 1+1=2}`, as the underlying SMT can always prove the equality `1+1=2`. We write `{1+1=2}` to simplify the type `{v:()| 1+1=2}` from the irrelevant binder `v:()`.

Program Properties as Types The theorems we express can refer to program functions. As an example, the type of `assoc` expresses that $\diamondsuit$ is associative.

```
assoc :: x:m → y:m → z:m → {x ◇ (y ◇ z) = (x ◇ y) ◇ z}
```

In § 6.1 we explain how to write Haskell proof terms to prove theorems like `assoc` by proving that list append `(++)` is associative. Moreover, we prove that the empty list `[]` is the identity element of list append and conclude that the list (with `[]` and `(++)`, *i.e.* the triple (`[a]`, `[]`, `(++)`)) is provably a monoid.

Corectness of Parallelization In § 6.2, we define the type `Morphism n m f` that specifies that `f` is a *morphism* between two monoids (`n`, η, $\boxdot$) and (`m`, ε, $\diamondsuit$), *i.e.* `f :: n → m` where `f` η `= ` ε and `f (x ⊡ y) = f x ◇ f y`.

A morphism `f` on a "chunkable" input type can be parallelized by:

- chunking up the input in `j` chunks (`chunk j`),
- applying the morphism in parallel to all chunks (`pmap f`), and
- recombining the mapped chunks via $\diamondsuit$, also in parallel (`pmconcat i`).

We specify correctness of the above transformation as a refinement type.

```
parallelismEq
```

```
:: f:(n → m) → Morphism n m f → x:n → i:Pos → j:Pos
→ {f x = pmconcat i (pmap f (chunk j x))}
```

§ 6.2 describes the parallelization transformation in details concluding with Correctness of Parallelization Theorem 10 that proves correctness by a Haskell definition of `parallelismEq` that satisfies the above type.

Case Study: Parallelization of String Matching We use the above theorem to parallelize string matching. We define a string matching function `toSM :: RString → toSM target` from a *refined string* to a *string matcher*. A refined string (§ 6.3.1) is a wrapper around the efficient string manipulation library `ByteString` that moreover assumes various string properties, including the monoid laws. A string matcher `SM target` (§ 6.3.2) is a data type that contains a refined string and a list of all the indices where the type level symbol `target` appears in the input. We prove that `SM target` is a monoid and `toSM` is a morphism, thus by the aforementioned Correctness of Parallelization Theorem 10 we can correctly parallelize string matching.

To sum up, we present the first realistic proof that uses Haskell as a theorem prover: correctness of parallelization on string matching. This chapter is summarized as follows

- We explain how theorems and proofs are encoded and machine checked in LIQUID HASKELL by formalizing monoids and proving that lists are monoids (§ 6.1).
- We formalize morphisms between monoids and specify and prove correctness of parallelization of morphisms (§ 6.2).
- We show how libraries can be imported as trusted components by wrapping `ByteString`s as refined strings which satisfy the monoid laws (§ 6.3.1).
- As an application, we prove that a string matcher is a morphism between the monoids of refined strings and string matchers, thus we get provably correct parallelization of string matching (§ 6.3).
- Based on our approximately 2K lines of code proof we evaluate the approach of using Haskell as a theorem prover (§ 6.4).

6.1 Proofs as Haskell Functions

Refinement Reflection, as explained in chapter 5, is a technique that lets you write Haskell functions that prove theorems about other Haskell functions and have your proofs machine-checked by LIQUID HASKELL. As an introduction to Refinement Reflection, in this section, we prove that lists are monoids by

- *specifying monoid laws* as refinement types,
- *proving the laws* by writing the implementation of the law specifications, and
- *verifying the proofs* using LIQUID HASKELL.

6.1.1 Reflection of data types into logic.

To start with, we define a List data structure and teach LIQUID HASKELL basic properties about List, namely, how to check that proofs on lists are *total* and how to encode functions on List into the logic.

The data list definition `L` is the standard recursive definition.

```
data L [length] a = N | C a (L a)
```

With the `length` annotation in the definition LIQUID HASKELL will use the `length` function to check termination of functions recursive on Lists. We define `length` as the standard Haskell function that returns natural numbers. We lift `length` into logic as a *measure* (§ 2.1.6), that is, a *unary* function whose (1) domain is the data type and (2) body is a single case-expression over the datatype.

```
type Nat = {v:Int | 0 ≤ v}


measure length    :: L a → Nat
  length N        = 0
  length (C x xs) = 1 + length xs
```

Finally, we teach LIQUID HASKELL how to encode functions on Lists into logic. The flag `"--exact-data-cons"` automatically derives measures which (1) test if a value has a given

data constructor and (2) extract the corresponding field's value. For example, LIQUID HASKELL will automatically derive the following List manipulation measures from the List definition.

```
isN :: L a → Bool    -- Haskell's null
isC :: L a → Bool    -- Haskell's not . null

selC1 :: L a → a     -- Haskell's head
selC2 :: L a → L a   -- Haskell's tail
```

Next, we describe how LIQUID HASKELL uses the above measures to automatically reflect Haskell functions on Lists into logic.

6.1.2 Reflection of Haskell functions into logic.

Next, we define and reflect into logic the two monoid operators on Lists. Namely, the identity element ε (which is the empty list) and an associative operator (◇) (which is list append).

```
reflect ε
ε :: L a
ε = N

reflect (◇)
(◇) :: L a → L a → L a
N        ◇ ys = ys
(C x xs) ◇ ys = C x (xs ◇ ys)
```

The reflect annotations lift the Haskell functions into logic in three steps. First, check that the Haskell functions indeed terminate by checking that the `length` of the input list is decreasing, as specified in the data list definition. Second, in the logic, they define the respective uninterpreted functions ε and (◇). Finally, the Haskell functions and the logical uninterpreted functions are related by strengthening the result type of the Haskell function with the definition of the function's implementation. For example, with the above `reflect` annotations, LIQUID HASKELL will *automatically* derive the following strengthened types for the relevant functions.

```
ε  :: {v:L a | v = ε ∧ v = N }
```

```
(◇) :: xs:L a → ys:L a
    → {v:L a | v = xs ◇ ys
             ∧ v = if isN xs then ys
                   else C (selC1 xs) (selC2 xs ◇ ys)
       }
```

6.1.3 Specification and Verification of Monoid Laws

Now we are ready to specify the monoid laws as refinement types and provide their respective proofs as terms of those type. LIQUID HASKELL will verify that our proofs are valid. Note that this is exactly what one would do in any standard logical framework, like LF [38].

The type `Proof` is predefined as an alias of the unit type (`()`) in the LIQUID HASKELL's library `ProofCombinators`. We summarize all the definitions we use from `ProofCombinators` in Figure 6.1. We express theorems as refinement types by refining the `Proof` type with appropriate refinements. For example, the following theorem states the ε is always equal to itself.

```
trivial :: {ε = ε}
```

Where `{ε = ε}` is a simplification for the `Proof` type `{v:Proof | ε = ε}`, since the binder `v` is irrelevant, and `trivial` is defined in `ProofCombinators` to be unit. LIQUID HASKELL will typecheck the above code using an SMT solver to check congruence on ε.

Definition 1 (Monoid)**.** *The triple (m, ε, $\diamondsuit$) is a monoid (with identity element ε and associative operator $\diamondsuit$), if the following functions are defined.*

$$idLeft_m :: x{:}m \rightarrow \{\varepsilon \diamondsuit x = x\}$$
$$idRight_m :: x{:}m \rightarrow \{x \diamondsuit \varepsilon = x\}$$
$$assoc_m :: x{:}m \rightarrow y{:}m \rightarrow z{:}m \rightarrow \{x \diamondsuit (y \diamondsuit z) = (x \diamondsuit y) \diamondsuit z\}$$

Using the above definition, we prove that our list type `L` is a monoid by defining Haskell proof terms that satisfy the above monoid laws.

Left Identity is expressed as a refinement type signature that takes as input a list `x:L a` and returns a `Proof` type refined with the property `ε ◇ x = x`

```
type Proof = ()
data QED   = QED

trivial :: Proof
trivial = ()

(=.) :: x:a → y:{a | x = y} → {v:a | v = x}
x =. _ = x

(**) :: a → QED → Proof
_ ** _ = ()

(∴) :: (Proof → a) → Proof → a
f ∴ y = f y
```

Figure 6.1. Operators and Types defined in `ProofCombinators`.

```
idLeft :: x:L a → {ε ♢ x = x}

idLeft x
  =  ε ♢ x
  =. N ♢ x
  =. x
  ** QED
```

We prove left identity using combinators from `ProofCombinators` as defined in Figure 6.1. We start from the left hand side `ε ♢ x`, which is equal to `N ♢ x` by calling `ε` thus unfolding the equality `ε = N` into the logic. Next, the call `N ♢ x` unfolds into the logic the definition of `(♢)` on `N` and `x`, which is equal to `x`, concluding our proof. Finally, we use the operators `p ** QED` which casts `p` into a proof term. In short, the proof of left identity, proceeds by unfolding the definitions of `ε` and `(♢)` on the empty list.

Right identity is proved by structural induction. We encode inductive proofs by case splitting on the base and inductive case and enforcing the inductive hypothesis via a recursive call.

```
idRight :: x:L a → { x ♢ ε = x }

idRight N
  =  N ♢ empty
  =. N
  ** QED
```

```
idRight (C x xs)
  =  (C x xs) ◇ empty
  =. C x (xs ◇ empty)
  =. C x xs ∴ idRight xs
  ** QED
```

The recursive call `idRight xs` is provided as a third optional argument in the `(=.)` operator to justify the equality `xs ◇ empty = xs`, while the operator `(∴)` is merely a function application with the appropriate precedence. Note that LiquiHaskell, via termination and totality checking, is verifying that all the proof terms are well formed because (1) the inductive hypothesis is only applying to smaller terms and (2) all cases are covered.

Associativity is proved in a very similar manner, using structural induction.

```
assoc :: x:L a → y:L a → z:L a → {x ◇ (y ◇ z) = (x ◇ y) ◇ z}
assoc N y z
  =  N ◇ (y ◇ z)
  =. y ◇ z
  =. (N ◇ y) ◇ z
  ** QED


assoc (C x xs) y z
  =  (C x xs) ◇ (y ◇ z)
  =. C x (xs ◇ (y ◇ z))
  =. C x ((xs ◇ y) ◇ z) ∴ associativity xs y z
  =. (C x (xs ◇ y)) ◇ z
  =. ((C x xs) ◇ y) ◇ z
  ** QED
```

As with the left identity, the proof proceeds by (1) function unfolding (or rewriting in paper and pencil proof terms), (2) case splitting (or case analysis), and (3) recursion (or induction).

Since our list implementation satisfies the three monoid laws we can conclude that `L a` is a monoid.

Theorem 7. *(L a, ε, ◇) is a monoid.*

Proof. `L a` is a monoid, as the implementation of `idLeft`, `idRight`, and `assoc` satisfy the specifications of $\texttt{idLeft}_m$, $\texttt{idRight}_m$, and $\texttt{assoc}_m$, with `m = L a`. □

6.2 Verified Parallelization of Monoid Morphisms

A monoid morphism is a function between two monoids which preserves the monoidal structure; *i.e.* a function on the underlying sets which preserves identity and associativity. We formally specify this definition using a refinement type `Morphism`.

Definition 2 (Monoid Morphism)**.** *A function* $f :: n \to m$ *is a morphism between the monoids* $(m, \varepsilon, \diamondsuit)$ *and* $(n, \eta, \boxdot)$*, if* $\texttt{Morphism}\ n\ m\ f$ *has an inhabitant.*

```
type Morphism n m F
  = x:n → y:n → {F η = ε ∧ F (x ⊡ y) = F x ♢ F y}
```

A monoid morphism can be parallelized when its domain can be cut into chunks and put back together again, a property we refer to as chunkable and expand upon in § 6.2.1. A chunkable monoid morphism is then parallelized by:

- chunking up the input,
- applying the morphism in parallel to all chunks, and
- recombining the chunks, also in parallel, back to a single value.

In the rest of this section we implement and verify to be correct the above transformation.

6.2.1 Chunkable Monoids

Definition 3 (Chunkable Monoids)**.** *A monoid* $(m, \varepsilon, \diamondsuit)$ *is chunkable if the following four functions are defined on* m*.*

```
length_m :: m → Nat

drop_m   :: i:Nat → x:MGEq m i → MEq m (length_m x - i)
take_m   :: i:Nat → x:MGEq m i → MEq m i
```

```
takeDropProp_m :: i:Nat → x:m → {x = take_m i x ◇ drop_m i x}
```

Where the type aliases `MLeq m I` *(and* `MEq m I`*) constrain the monoid* `m` *to have* $length_m$ *greater than (resp. equal) to* `I`*.*

```
type MGEq m I = {x:m | I ≤ length_m x}
type MEq  m I = {x:m | I = length_m x}
```

Note that the "important" methods of chunkable monoids are the `take` and `drop`, while the `length` method is required to give pre- and post-condition on the other operations. Finally, `takeDropProp` provides a proof that for each `i` and monoid `x`, appending `take i x` to `drop i x` will reconstruct `x`.

Using $take_m$ and $drop_m$ we define for each chunkable monoid (`m`, ε, $\diamondsuit$) a function $chunk_m$ `i x` that splits `x` in chunks of size `i`.

```
chunk_m :: i:Pos → x:m → {v:L m | chunkRes_m i x v}
chunk_m i x
  | length_m x ≤ i = C x N
  | otherwise      = take_m i x `C` chunk_m i (drop_m i x)

chunkRes_m i x v
  | length_m x ≤ i = length_m v == 1
  | i == 1         = length_m v == length_m xs
  | otherwise      = length_m v <  length_m xs
```

The function $chunk_m$ provably terminates as $drop_m$ `i x` will return a monoid smaller than `x`, by the Definition of $drop_m$. The definitions of both $take_m$ and $drop_m$ are also used from Liquid Haskell to verify the $length_m$ constraints in the result of $chunk_m$.

6.2.2 Parallel Map

We define a parallelized map function `pmap` using Haskell's library `parallel`. Concretely, we use the function `Control.Parallel.Strategies.withStrategy` that computes its argument in parallel given a parallel strategy.

```
pmap :: (a → b) → L a → L b
pmap f xs = withStrategy parStrategy (map f xs)
```

The strategy `parStrategy` does not affect verification. In our codebase we choose the traversable strategy.

```
parStrategy :: Strategy (L a)
parStrategy = parTraversable rseq
```

Parallelism in the Logic The function `withStrategy` is an imported Haskell library function, whose implementation is not available during verification. To use it in our verified code, we make the *assumption* that it always returns its second argument.

```
assume withStrategy :: Strategy a → x:a → {v:a | v = x}
```

Moreover, we need to reflect the function `pmap` and represent its implementation in the logic. Thus, we also need to represent the function `withStrategy` in the logic. LiquidHaskell represents `withStrategy` in the logic as a logical function that merely returns its second argument, `withStrategy _ x = x`, and does not reason about parallelism.

6.2.3 Monoidal Concatenation

The function $\texttt{chunk}_m$ allows chunking a monoidal value into several pieces. Dually, for any monoid `m`, there is a standard way of turning `L m` back into a single `m` [1].

```
mconcat :: L m → m
mconcat N        = ε
mconcat (C x xs) = x ◇ mconcat xs
```

For any chunkable monoid `n`, monoid morphism `f :: n → m`, and natural number `i > 0` we can write a chunked version of `f` as

$$\texttt{mconcat . pmap f . chunk}_n\ \texttt{i :: n → m.}$$

Before parallelizing `mconcat`, we will prove that the previous function is equivalent to `f`.

[1] `mconcat` is usually defined as `foldr mappend mempty`

Theorem 8 (Morphism Distribution)**.** *Let (m, ε, ♢) be a monoid and (n, η, ⊡) be a chunkable monoid. Then, for every morphism* `f :: n → m`*, every positive number* `i` *and input* `x`*,* `f x = mconcat (pmap f (chunk`$_n$ `i x))` *holds.*

```
morphismDistribution
  :: f:(n → m) → Morphism n m f → x:n → i:Pos
  → {f x = mconcat (pmap f (chunk_n i x))}
```

Proof. We prove the theorem by implementing `morphismDistribution` in a way that satisfies its type. The proof proceeds by induction on the length of the input.

```
morphismDistribution f thm x i
  | length_n x ≤ i
  =  mconcat (pmap f (chunk_n i x))
  =. mconcat (map f (chunk_n i x))
  =. mconcat (map f (C x N))
  =. mconcat (f x `C` map f N)
  =. f is ♢ mconcat N
  =. f is ♢ ε
  =. f is ∴ idRight_m (f is)
  ** QED

morphismDistribution f thm x i
  =   mconcat (pmap f (chunk_n i x))
  =. mconcat (map f (chunk_n i x))
  =. mconcat (map f (C takeX) (chunk_n i dropX)))
  =. mconcat (f takeX `C` map f (chunk_n n dropX))
  =. f takeX ♢ f dropX   ∴ morphismDistribution f thm dropX i
  =. f (takeX ⊡ dropX)   ∴ thm takeX dropX
  =. f x                 ∴ takeDropProp_n i x
  ** QED
  where
    dropX = drop_n i x
    takeX = take_n i x
```

In the base case we use rewriting and right identity on the monoid `f x`. In the inductive case, we

use the inductive hypothesis on the input `dropX = drop`$_n$ `i x`, that is provably smaller than `x` as `1 < i`. Then, the fact that `f` is a monoid morphism, as encoded by our assumption argument `thm takeX dropX` we get basic distribution of `f`, that is `f takeX ♢ f dropX = f (takeX ⊡ dropX)`. Finally, we merge `takeX ⊡ dropX` to `x` using the property `takeDropProp`$_n$ of the chunkable monoid `n`. □

6.2.4 Parallel Monoidal Concatenation

We now parallelize the monoid concatenation by defining a `pmconat i x` function that chunks the input list of monoids and concatenates each chunk in parallel.

We use the `chunk` function of § 6.2.1 instantiated to `L m` to define a parallelized version of monoid concatenation `pmconcat`.

```
pmconcat :: Int → L m → m
pmconcat i x | i ≤ 1 || length x ≤ i
  = mconcat x
pmconcat i x
  = pmconcat i (pmap mconcat (chunk i x))
```

The function `pmconcat i x` calls `mconcat x` in the base case, otherwise it (1) chunks the list `x` in lists of size `i`, (2) runs in parallel `mconcat` to each chunk, (3) recursively runs itself with the resulting list. Termination of `pmconcat` holds, as the length of `chunk i x` is smaller than the length of `x`, when `1 < i`.

Next, we prove equivalence of parallelized monoid concatenation.

Theorem 9 (Correctness of Parallelization)**.** *Let (m, ε, ♢) be a monoid. Then, the parallel and sequential concatenations are equivalent.*

```
pmconcatEq :: i:Int → x:L m → {pmconcat i x = mconcat x}
```

Proof. We prove the theorem by providing a Haskell implementation of `pmconcatEq` that satisfies its type. The details of the proof can be found in [97], here we provide the sketch of the proof.

First, we prove that `mconcat` distributes over list splitting

```
mconcatSplit
```

```
  :: i:Nat → xs:{L m | i ≤ length xs}
  → {mconcat xs = mconcat (take i xs) ♢ mconcat (drop i xs)}
```

The proofs proceeds by structural induction, using monoid left identity in the base case and monoid associativity associavity and unfolding of `take` and `drop` methods in the inductive step.

We generalize the above to prove that `mconcat` distributes over list chunking.

```
mconcatChunk
  :: i:Pos → xs:L m
  → {mconcat xs = mconcat (map mconcat (chunk i xs))}
```

The proofs proceeds by structural induction, using monoid left identity in the base case and lemma `mconcatSplit` in the inductive step.

Lemma `mconcatChunk` is sufficient to prove `pmconcatEq` by structural induction, using monoid left identity in the base case. □

6.2.5 Parallel Monoid Morphism

We can now replace the `mconcat` in our chunked monoid morphism in § 6.2.3 with `pmconcat` from § 6.2.4 to provide an implementation that uses parallelism to both map the monoid morphism and concatenate the results.

Theorem 10 (Correctness of Parallelization)**.** *Let $(m, \varepsilon, \diamondsuit)$ be a monoid and $(n, \eta, \boxdot)$ be a chunkable monoid. Then, for every morphism `f :: n → m`, every positive numbers `i` and `j`, and input `x`, `f x = pmconcat i (pmap f (chunk`$_n$ `j x))` holds.*

```
parallelismEq
  :: f:(n → m) → Morphism n m f → x:n → i:Pos → j:Pos
  → {f x = pmconcat i (pmap f (chunk_n j x))}
```

Proof. We prove the theorem by providing an implementation of `parallelismEq` that satisfies its type.

```
parallelismEq f thm x i j
  =   pmconcat i (pmap f (chunk_n j x))
  =. mconcat (pmap f (chunk_n j x))
```

```
   ∴ pmconcatEq i (pmap f (chunk_n j x))
=. f x
   ∴ morphismDistribution f thm x j
** QED
```

The proof follows merely by application of the two previous Theorems 8 and 9. □

A Basic Time Complexity analysis of the algorithm reveals that parallelization of morphism leads to runtime speedups on monads with fast (constant time) appending operator.

We want to compare the complexities of the sequential `f i` and the two-level parallel `pmconcat i (pmap f (chunk`$_n$ `j x))`. Let n be the size on the input `x`. Then, the sequential version runs in time $T_f(n) = O(n)$, that is equal to the time complexity of the morphism `f` on input `i`.

The parallel version runs `f` on inputs of size $n' = \frac{n}{j}$. Assuming the complexity of `x` $\diamondsuit$ `y` to be $T_\diamondsuit(\max(|\mathtt{x}|,|\mathtt{y}|))$, complexity of `mconcat xs` is $O((\mathtt{length\ xs}-1)T_\diamondsuit(\max_{\mathtt{x}_i\in\mathtt{xs}}(|\mathtt{x}_i|)))$. Now, parallel concatenation, `pmconcat i xs` at each iteration runs $\diamondsuit$ on a list of size `i`. Moreover, at each iteration, divides the input list in chunks of size `i`, leading to $\frac{\log|xs|}{\log i}$ iterations, and time complexity $(i-1)(\frac{\log|xs|}{\log i})(T_\diamondsuit(m))$ for some m that bounds the size of the monoids.

The time complexity of parallel algorithm consists on the base cost on running `f` at each chunk and then parallel concatenating the $\frac{n}{j}$ chunks.

$$O((i-1)(\frac{\log n - \log j}{\log i})T_\diamondsuit(m) + T_f(\frac{n}{j})) \tag{6.1}$$

Since time complexity depends on the time complexity of $\diamondsuit$ for the parallel algorithm to be efficient time complexity of $\diamondsuit$ should be constant. Otherwise, if it depends on the size of the input, the size of monoids can grow at each iteration of `mconcat`.

Moreover, from the complexity analysis we observe that time grows on bigger `i` and smaller `j`. Thus, chunking the input in small chunks while splitting the monoid list in half leads to more parallelism, and thus (assuming infinite processors and no caching) greatest speedup.

6.3 Correctness of Parallel String Matching

In § 6.2 we showed that any monoid morphism whose domain is chunkable can be parallelized. We now apply that result to parallelize string matching. We start by observing that strings are a chunkable monoid. We then turn string matching for a given target into a monoid morphism from a string to a suitable monoid, `SM target`, defined in § 6.3.2. Finally, in § 6.3.4, we parallelize string matching by a simple use of the parallel morphism function of § 6.2.5.

6.3.1 Refined Strings are Chunkable Monoids

We define a new type `RString`, which is a chunkable monoid, to be the domain of our string matching function. Our type simply wraps Haskell's existing `ByteString`.

```
data RString = RS BS.ByteString
```

Similarly, we wrap the existing `ByteString` functions we will need to show `RString` is a chunkable monoid.

```
η = RS (BS.empty)
(RS x) ⊡ (RS y)= S (x `BS.append` y)

lenStr    (RS x) = BS.length x
takeStr i (RS x) = RS (BS.take i x)
dropStr i (RS x) = RS (BS.take i x)
```

Although it is possible to explicitly prove that `ByteString` implements a chunkable monoid [99], it is time consuming and orthogonal to our purpose. Instead, we just *assume* the chunkable monoid properties of `RString`– thus demonstrating that refinement reflection is capable of doing gradual verification.

For instance, we define a logical uninterpreted function ⊡ and relate it to the Haskell ⊡ function via an assumed (unchecked) type.

```
assume (⊡) :: x:RString → y:RString → {v:RString | v = x ⊡ y}
```

Then, we use the uninterpreted function ⊡ in the logic to assume monoid laws, like associativity.

```
assume assocStr
```

```
    :: x:RString → y:RString → z:RString
    → {x ⊡ (y ⊡ z) = (x ⊡ y) ⊡ z}
assocStr _ _     = trivial
```

Haskell applications of ⊡ are interpreted in the logic via the logical ⊡ that satisfies associativity via theorem `assocStr`.

Similarly for the chunkable methods, we define the uninterpreted functions `takeStr`, `dropStr` and `lenStr` in the logic, and use them to strengthen the result types of the respective functions. With the above function definitions (in both Haskell and logic) and assumed type specifications, Liquid Haskell will check (or rather assume) that the specifications of chunkable monoid, as defined in the Definitions 1 and 3, are satisfied. We conclude with the assumption (rather that theorem) that `RString` is a chunkable monoid.

Assumption 11 (RString is a Chunkable Monoid)**.** *(`RString`, η, ⊡) combined with the methods `lenStr`, `takeStr`, `dropStr` and `takeDropPropStr` is a chunkable monoid.*

6.3.2 String Matching Monoid

String matching is determining all the indices in a source string where a given target string begins; for example, for source string `ababab` and target `aba` the results of string matching would be `[0, 2]`.

We now define a suitable monoid, `SM target`, for the codomain of a string matching function, where `target` is the string being looked for. Additionally, we will define a function `toSM :: RString → SM target` which does the string matching and is indeed a monoid morphism from `RString` to `SM target` for a given `target`.

String Matching Monoid

We define the data type `SM target` to contain a refined string field `input` and a list of all the indices in `input` where the `target` appears.

```
data SM (target :: Symbol) where
  SM :: input:RString
     → indices:[GoodIndex input target]
```

```
      → SM target
```

We use the string type literal [2] to parameterize the monoid over the target being matched. This encoding allows the type checker to statically ensure that only searches for the same target can be merged together. The input field is a refined string, and the indices field is a list of good indices. For simplicity we present lists as Haskell's built-in lists, but our implementation uses the reflected list type, `L`, defined in § 6.1.

A `GoodIndex input target` is a refined type alias for a natural number `i` for which `target` appears at position `i` of `input`. As an example, the good indices of `"abcab"` on `"ababcabcab"` are `[2,5]`.

```
type GoodIndex Input Target
  = {i:Nat | isGoodIndex Input (fromString Target) i }

isGoodIndex :: RString → RString → Int → Bool
isGoodIndex input target i
  = (subString i (lenStr target) input  == target)
  ∧ (i + lenStr target ≤ lenStr input)

subString :: Int → Int → RString → RString
subString o l = takeStr l . dropStr o
```

Monoid Methods for String Matching

Next, we define the mappend and identity elements for string matching.

The *identity element* ε of `SM t`, for each target `t`, is defined to contain the identity `RString` (η) and the identity `List` (`[]`).

```
ε:: ∀ (t :: Symbol). SM t
ε = SM η []
```

The Haskell definition of $\diamondsuit$, the monoid operation for `SM t`, is as follows.

```
(♢)::∀ (t::Symbol). KnownSymbol t ⇒ SM t → SM t → SM t
(SM x xis) ♢ (SM y yis)
```

[2] `Symbol` is a kind and target is effectively a singleton type.

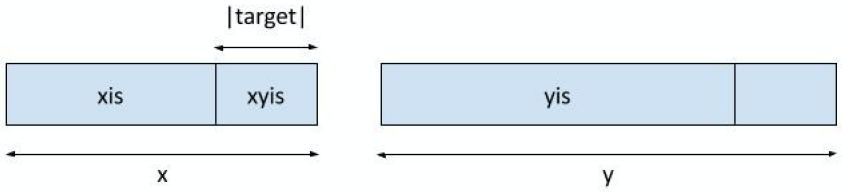

Figure 6.2. Mappend indices of String Matcher.

```
= SM (x ⊡ y) (xis′ ++ xyis ++ yis′)
where
  tg   = fromString (symbolVal (Proxy :: Proxy t))
  xis′ = map (castGoodIndexLeft tg x y) xis
  xyis = makeNewIndices x y tg
  yis′ = map (shiftStringRight tg x y) yis
```

Note again that capturing target as a type parameter is critical, otherwise there is no way for the Haskell's type system to specify that both arguments of (◇) are string matchers on the same target.

The action of (◇) on the two `input` fields is straightforward; however, the action on the two `indices` is complicated by the need to shift indices and the possibility of new matches arising from the concatenation of the two `input` fields. Figure 6.2 illustrates the three pieces of the new `indices` field which we now explain in more detail.

1. Casting Good Indices If `xis` is a list of good indices for the string `x` and the target `tg`, then `xis` is also a list of good indices for the string `x ⊡ y` and the target `tg`, for each `y`. To prove this property we need to invoke the property `subStrAppendRight` on Refined Strings that establishes substring preservation on string right appending.

```
assume subStrAppendRight
    :: sl:RString → sr:RString → j:Int
    → i:{Int | i + j ≤ lenStr sl }
    → {subString sl i j = subString (sl ⊡ sr) i j}
```

The specification of `subStrAppendRight` ensures that for each string `sl` and `sr` and each integer `i` and `j` whose sum is within `sl`, the substring from `i` with length `j` is identical in `sl` and in `(sl ⊡ sr)`. The function `castGoodIndexLeft` applies the above property to an index `i` to cast

it from a good index on `sl` to a good index on `(sl ⊡ sr)`

```
castGoodIndexLeft
  :: tg:RString → sl:RString → sr:RString
  → i:GoodIndex sl tg
  → {v:GoodIndex (sl ⊡ sr) target | v = i}

castGoodIndexLeft tg sl sr i
  = cast (subStrAppendRight sl sr (lenStr tg) i) i
```

Where `cast p x` returns `x`, after enforcing the properties of `p` in the logic

```
cast :: b → x:a → {v:a | v = x}
cast _ x = x
```

Moreover, in the logic, each expression `cast p x` is reflected as `x`, thus allowing random (*i.e.* non-reflected) Haskell expressions to appear in `p`.

2. Creation of new indices The concatenation of two input strings `sl` and `sr` may create new good indices. For instance, concatenation of `"ababcab"` with `"cab"` leads to a new occurence of `"abcab"` at index `5` which does not occur in either of the two input strings. These new good indices can appear only at the last `lenStr tg` positions of the left input `sl`. `makeNewIndices sl sr tg` detects all such good new indices.

```
makeNewIndices
  :: sl:RString → sr:RString → tg:RString
  → [GoodIndex {sl ⊡ sr} tg]

makeNewIndices sl sr tg
  | lenStr tg < 2 = []
  | otherwise     = makeIndices (sl ⊡ sr) tg lo hi
  where
    lo = maxInt (lenStr sl - (lenStr tg - 1)) 0
    hi = lenStr sl - 1
```

If the length of the `tg` is less than 2, then no new good indices are created. Otherwise, the call on `makeIndices` returns all the good indices of the input `sl ⊡ sr` for target `tg` in the range from

`maxInt (lenStr sl-(lenStr tg-1)) 0` to `lenStr sl-1`.

Generally, `makeIndices s tg lo hi` returns the good indices of the input string `s` for target `tg` in the range from `lo` to `hi`.

```
makeIndices :: s:RString → tg:RString → lo:Nat
            → hi:Int → [GoodIndex s tg]

makeIndices s tg lo hi
  | hi < lo              = []
  | isGoodIndex s tg lo = lo:rest
  | otherwise            = rest
  where
    rest = makeIndices s tg (lo + 1) hi
```

It is important to note that `makeNewIndices` does not scan all the input, instead only searching at most `lenStr tg` positions for new good indices. Thus, the time complexity to create the new indices is linear on the size of the target but independent of the size of the input.

3. Shift Good Indices If `yis` is a list of good indices on the string `y` with target `tg`, then we need to shift each element of `yis` right `lenStr x` units to get a list of good indices for the string `x ⊡ y`.

To prove this property we need to invoke the property `subStrAppendLeft` on Refined Strings that establishes substring shifting on string left appending.

```
assume subStrAppendLeft
  :: sl:RString → sr:RString
  → j:Int → i:Int
  → {subStr sr i j = subStr (sl ⊡ sr) (lenStr sl+i) j}
```

The specification of `subStrAppendLeft` ensures that for each string `sl` and `sr` and each integers `i` and `j`, the substring from `i` with length `j` on `sr` is equal to the substring from `lenStr sl + i` with length `j` on `(sl ⊡ sr)`. The function `shiftStringRight` both shifts the input index `i` by `lenStr sl` and applies the `subStrAppendLeft` property to it, casting `i` from a good index on `sr` to a good index on `(sl ⊡ sr)`

Thus, `shiftStringRight` both appropriately shifts the index and casts the shifted index using the above theorem:

```
shiftStringRight
  :: tg:RString → sl:RString → sr:RString
  → i:GoodIndex sr tg
  → {v:(GoodIndex (sl ⊡ sr) tg) | v = i + lenStr sl}

shiftStringRight tg sl sr i
  = subStrAppendLeft sl sr (lenStr tg) i `cast` i + lenStr sl
```

String Matching is a Monoid

Next we prove that the monoid methods ε and $(\diamondsuit)$ satisfy the monoid laws.

Theorem 12 (SM is a Monoid)**.** *(SM t, ε, ♢) is a monoid.*

Proof. According to the Monoid Definition 1, we prove that string matching is a monoid, by providing safe implementations for the monoid law functions. First, we prove *left identity*.

```
idLeft :: x:SM t → {ε ♢ x = xs}

idLeft (SM i is)
  =   (ε :: SM t) ♢ (SM i is)
  =. (SM η []) ♢ (SM i is)
  =. SM (η ⊡ i) (is1 ++ isNew ++ is2)
     ∴ idLeftStr i
  =. SM i ([] ++ [] ++ is)
     ∴ (mapShiftZero tg i is ∧ newIsNullRight i tg)
  =. SM i is
     ∴ idLeftList is
  ** QED
  where
    tg    = fromString (symbolVal (Proxy :: Proxy t))
    is1   = map (castGoodIndexRight tg i η) []
    isNew = makeNewIndices η i tg
    is2   = map (shiftStringRight tg η i) is
```

The proof proceeds by rewriting, using left identity of the monoid strings and lists, and two more lemmata.

- Identity of shifting by an empty string.

```
mapShiftZero :: tg:RString → i:RString
  → is:[GoodIndex i target]
  → {map (shiftStringRight tg η i) is = is}
```

  The lemma is proven by induction on `is` and the assumption that empty strings have length 0.

- No new indices are created.

```
newIsNullLeft :: s:RString → t:RString
              → {makeNewIndices η s t = []}
```

  The proof relies on the fact that `makeIndices` is called on the empty range from `0` to `-1` and returns `[]`.

Next, we prove *right identity*.

```
idRight :: x:SM t → {x ♢ ε = x}

idRight (SM i is)
  =  (SM i is) ♢ (ε :: SM t)
  =. (SM i is) ♢ (SM η [])
  =. SM (i ⊡ η) (is1 ++ isNew ++ is2)
     ∴ idRightStr i
  =. SM i (is ++ N ++ N)
     ∴ (mapCastId tg i η is ∧ newIsNullLeft i tg)
  =. SM i is
     ∴ idRightList is
  **  QED
  where
    tg    = fromString (symbolVal (Proxy :: Proxy t))
    is1   = map (castGoodIndexRight tg i η) is
```

```
isNew = makeNewIndices i stringEmp tg
is2   = map (shiftStringRight tg i η) []
```

The proof proceeds by rewriting, using right identity on strings and lists and two more lemmata.

- Identity of casting is proven

  ```
  mapCastId
    :: tg:RString → x:RString → y:RString
    → is:[GoodIndex x tg] →
    → {map (castGoodIndexRight tg x y) is = is}
  ```

  We prove identity of casts by induction on `is` and identity of casting on a single index.

- No new indices are created.

  ```
  newIsNullLeft :: s:RString → t:RString
                → {makeNewIndices s η t = []}
  ```

  The proof proceeds by case splitting on the relative length of `s` and `t`. At each case we prove by induction that all the potential new indices would be out of bounds and thus no new good indices would be created.

- Finally we prove *associativity*. For space, we only provide a proof sketch. The whole proof is available online [97]. Our goal is to show equality of the left and right associative string matchers.

```
assoc :: x:SM t → y:SM t → z:SM t
      → {x ◇ (y ◇ z) = (x ◇ y) ◇ z}
```

To prove equality of the two string matchers we show that the input and indices fields are respectively equal. Equality of the input fields follows by associativity of RStrings. Equality of the index list proceeds in three steps.

1. Using list associativity and distribution of index shifting, we group the indices in the five lists shown in Figure 6.3: the indices of the input `x`, the new indices from mappending `x` to `y`, the indices of the input `y`, the new indices from mappending `x` to `y`, and the indices of the input `z`.

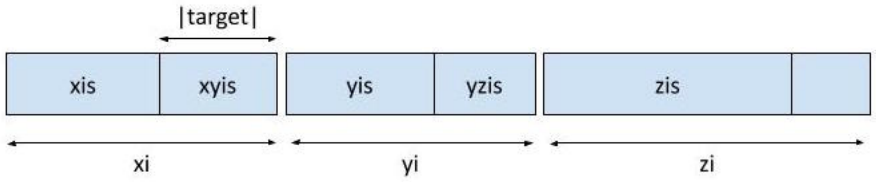

Figure 6.3. Associativity of String Matching.

2. The representation of each group depends on the order of appending. For example, if `zis1` (resp. `zis2`) is the group `zis` when right (resp. left) mappend happened first, then we have

```
zis1 = map (shiftStringRight tg xi (yi ⊡ zi))
           (map (shiftStringRight tg yi zi) zis)

zis2 = map (shiftStringRight tg (xi ⊡ yi) zi) zis
```

   That is, in right first, the indices of `z` are first shifted by the length of `yi` and then by the length of `xi`, while in the left first case, the indices of `z` are shifted by the length of `xi ⊡ yi`. In this second step of the proof we prove, using lemmata, the equivalence of the different group representations. The most interesting lemma we use is called `assocNewIndices` and proves equivalence of all the three middle groups together by case analysis on the relative lengths of the target `tg` and the middle string `yi`.

3. After proving equivalence of representations, we again use list associativity and distribution of casts to wrap the index groups back in string matchers.

We now sketch the three proof steps, while the whole proof is available online [97].

```
assoc x@(SM xi xis) y@(SM yi yis) z@(SM zi zis)
  -- Step 1: unwrapping the indices
  =   x ♢ (y ♢ z)
  =. (SM xi xis) ♢ ((SM yi yis) ♢ (SM zi zis))
                        ...
  -- via list associativity and distribution of shifts
  =. SM i (xis1 ++ ((xyis1 ++ yis1 ++ yzis1) ++ zis1))
  -- Step 2: Equivalence of representations
```

```
  =. SM i (xis2 ++ ((xyis1 ++ yis1 ++ yzis1) ++ zis1))
     ∴ castConcat tg xi yi zi xis
  =. SM i (xis2 ++ ((xyis1 ++ yis1 ++ yzis1) ++ zis2))
     ∴ mapLenFusion tg xi yi zi zis
  =. SM i (xis2 ++ ((xyis2 ++ yis2 ++ yzis2) ++ zis2))
     ∴ assocNewIndices y tg xi yi zi yis
  -- Step 3: Wrapping the indices
                        ...
  -- via list associativity and distribution of casts
  =. (SM xi xis ♢ SM yi yis) ♢ SM zi zis
  =. (x ♢ y) ♢ z
  ** QED
  where
    i     = xi ⊡ (yi ⊡ zi)

    yzis1 = map (shiftStringRight tg xi (yi ⊡ zi)) yzis
    yzis2 = makeNewIndices (xi ⊡ yi) zi tg
    yzis  = makeNewIndices yi zi tg
    ...
```

□

6.3.3 String Matching Monoid Morphism

Next, we define the function `toSM :: RString → SM target` which does the actual string matching computation for a set target [3]

```
toSM :: ∀ (target :: Symbol). (KnownSymbol target)
     ⇒ RString → SM target
toSM input = SM input (makeSMIndices input tg) where
  tg = fromString (symbolVal (Proxy :: Proxy target))

makeSMIndices
  :: x:RString → tg:RString → [GoodIndex x tg]
makeSMIndices x tg
```

[3] `toSM` assumes the target is clear from the calling context; it is also possible to write a wrapper function taking an explicit target which gets existentially reflected into the type.

```
  = makeIndices x tg 0 (lenStr tg - 1)
```

The input field of the result is the input string; the indices field is computed by calling the function `makeIndices` within the range of the `input`, that is from `0` to `lenStr input - 1`.

We now prove that `toSM` is a monoid morphism.

Theorem 13 (`toSM` is a Morphism)**.** *`toSM :: RString` $\rightarrow$ `SM t` is a morphism between the monoids (`RString`, η, $\boxdot$) and (`SM t`, ε, $\diamondsuit$).*

Proof. Based on definition 2, proving `toSM` is a morphism requires constructing a valid inhabitant of the type

```
type Morphism RString (SM t) toSM
  = x:RString → y:RString
  → {toSM η = ε ∧ toSM (x ⊡ y) = toSM x ♢ toSM y}
```

We define the function `distributestoSM :: Morphism RString (SM t) toSM` to be the required valid inhabitant.

The core of the proof starts from exploring the string matcher `toSM x ♢ toSM y`. This string matcher contains three sets of indices as illustrated in Figure 6.2: (1) `xis` from the input `x`, (2) `xyis` from appending the two strings, and (3) `yis` from the input `y`. We prove that appending these three groups of indices together gives exactly the good indices of `x ⊡ y`, which are also the value of the indices field in the result of `toSM (x ⊡ y)`.

```
distributestoSM x y
  =  (toSM x :: SM target) ♢ (toSM y :: SM target)
  =. (SM x is1) ♢ (SM y is2)
  =. SM i (xis ++ xyis ++ yis)
  =. SM i (makeIndices i tg 0 hi1 ++ yis)
     ∴ (mapCastId tg x y is1 ∧ mergeNewIndices tg x y)
  =. SM i (makeIndices i tg 0 hi1 ++ makeIndices i tg (hi1+1) hi)
     ∴ shiftIndicesRight 0 hi2 x y tg
  =. SM i is
     ∴ mergeIndices i tg 0 hi1 hi
  =. toSM (x ⊡ y)
```

```
  ** QED
  where
    xis  = map (castGoodIndexRight tg x y) is1
    xyis = makeNewIndices x y tg
    yis  = map (shiftStringRight   tg x y) is2
    tg   = fromString (symbolVal (Proxy::Proxy target))
    is1  = makeSMIndices x tg
    is2  = makeSMIndices y tg
    is   = makeSMIndices i tg
    i    = x ⊡ y
    hi1  = lenStr x - 1
    hi2  = lenStr y - 1
    hi   = lenStr i - 1
```

The most interesting lemma we use is `mergeIndices x tg lo mid hi` that states that for the input `x` and the target `tg` if we append the indices in the range from `to` to `mid` with the indices in the range from `mid+1` to `hi`, we get exactly the indices in the range from `lo` to `hi`. This property is formalized in the type of the lemma.

```
mergeIndices
  :: x:RString → tg:RString
  → lo:Nat → mid:{Int | lo ≤ mid} → hi:{Int | mid ≤ hi}
  → {   makeIndices x tg lo hi
    =  makeIndices x tg lo mid
    ++ makeIndices x tg (mid+1) hi}
```

The proof proceeds by induction on `mid` and using three more lemmata:

- `mergeNewIndices` states that appending the indices `xis` and `xyis` is equivalent to the good indices of `x ⊡ y` from `0` to `lenStr x - 1`. The proof case splits on the relative sizes of `tg` and `x` and is using `mergeIndices` on `mid = lenStr x1 - lenStr tg` in the case where `tg` is smaller than `x`.

- `mapCastId` states that casting a list of indices returns the same list.

- `shiftIndicesRight` states that shifting right `i` units the indices from `lo` to `hi` is equiva-

lent to computing the indices from `i + lo` to `i + hi` on the string `x ⊡ y`, with `lenStr x = i`.

□

6.3.4 Parallel String Matching

We conclude this section with the definition of a parallelized version of string matching. We put all the theorems together to prove that the sequential and parallel versions always give the same result.

We define `toSMPar` as a parallel version of `toSM` using machinery of section 6.2.

```
toSMPar :: ∀ (target :: Symbol). (KnownSymbol target)
        ⇒ Int → Int → RString → SM target
toSMPar i j = pmconcat i . pmap toSM . chunkStr j
```

First, `chunkStr` splits the input into `j` chunks. Then, `pmap` applies `toSM` at each chunk in parallel. Finally, `pmconat` concatenates the mapped chunks in parallel using $\diamondsuit$, the monoidal operation for `SM target`.

Correctness We prove correctness of `toSMPar` directly from Theorem 10.

Theorem 14 (Correctness of Parallel String Matching)**.** *For each parameter `i` and `j`, and input `x`, `toSMPar i j x` is always equal to `toSM x`.*

```
correctness :: i:Int → j:Int → x:RString
            → {toSM x = toSMPar i j x}
```

Proof. The proof follows by direct application of Theorem 10 on the chunkable monoid (`RString`, η, $\boxdot$) (by Assumption 11) and the monoid (`SM t`, ε, $\diamondsuit$) (by Theorem 12).

```
correctness i j x
  =   toSMPar i j x
  =. pmconcat i (pmap toSM (chunkStr j x))
  =. toSM is
     ∴ parallelismEq toSM distributestoSM x i j
  ** QED
```

Note that application of the theorem `parallelismEq` requires a proof that its first argument `toSM` is a morphism. By Theorem 8, the required proof is provided as the function `distributestoSM`.

□

Time Complexity Counting only string comparisons as the expensive operations, the sequential string matcher on input `x` runs in time linear to `n = lenStr x`. Thus $T_{\texttt{toSM}}(n) = O(n)$.

We get time complexity of `toSMPar` by the time complexity of two-level parallel algorithms equation 6.1, with the time of string matching mappend being linear on the length of the target `t = lenStr tg`, or $T_{\diamondsuit}(\texttt{SM}) = O(t)$.

$$T_{\texttt{toSMPar}}(n,t,i,j) = O((i-1)(\frac{\log n - \log j}{\log i})\, t + \frac{n}{j})$$

The above analysis refers to a model with infinite processor and no caching. To compare the algorithms in practice, we matched the target "the" in Oscar Wilde's "The Picture of Dorian Gray", a text of `n = 431372` characters using a two processor Intel Core i5. The sequential algorithm detected 4590 indices in 40 ms. We experimented with different parallization factors `i` and chunk sizes `j / n` and observed up to 50% speedups of the parallel algorithm for parallelization factor `4` and `8` chunks. As a different experiment, we matched the input against its size `t = 400` prefix, a size comparable to the input size `n`. For bigger targets, mappend gets slower, as it has complexity linear to the size of target. We observed 20% speedups for `t=400` target but also 30% slow downs for various sizes of `i` and `j`. In all cases the indices returned by the sequential and the parallel algorithms were the same.

6.4 Evaluation: Strengths & Limitations

Verification of Parallel String Matching is the first realistic proof that uses (Liquid) Haskell to prove properties *about* program functions. In this section we use the String Matching proof to quantitatively and qualitatively evaluate theorem proving in Haskell.

Quantitative Evaluation. The Correctness of Parallel String Matching proof can be found online [97]. Verification time, that is the time Liquid Haskell needs to check the proof, is 75 sec on a dual-core Intel Core i5-4278U processor. The proof consists of *1839* lines of code. Out of those

- *226* are Haskell "runtime" code,
- *112* are liquid comments on the "runtime" Haskell code,
- *1307* are Haskell proof terms, that is functions with `Proof` result type, and
- *194* are liquid comments to specify theorems.

Counting both liquid comments and Haskell proof terms as verification code, we conclude that the proof requires 7x the lines of "runtime" code. This ratio is high and takes us to 2006 Coq, when Leroy [58] verified the initial CompCert C compiler with the ratio of verification to compiler lines being 6x.

Strengths. Though currently verbose, deep verification using Liquid Haskell has many benefits. First and foremost, *the target code is written in the general purpose Haskell* and thus can use advanced Haskell features, including type literals, deriving instances, inline annotations and optimized library functions like `ByteString`. Even diverging functions can coexist with the target code, as long as they are not reflected into logic [100].

Moreover, *SMTs are used to automate the proofs* over key theories like linear arithmetic and equality. As an example, associativity of `(+)` is assumed throughout the proofs while shifting indices. Our proof could be further automated by mapping refined strings to SMT strings and using the automated SMT string theory. We did not follow this approach because we want to show that our techinique can be used to prove any (and not only domain specific) program properties.

Finally, we get further automation via *Liquid Type Inference* [79]. Properties about program functions, expressed as type specifications with unit result, often depend on program invariants, expressed as vanilla refinement types, and vice versa. For example, we need the invariant that all indices of a string matcher are good indices to prove associativity of $(\diamondsuit)$. Even

though Liquid Haskell cannot currently synthesize proof terms, it performs really well at inferring and propagating program invariants (like good indices) via the abstract interpretation framework of Liquid Types.

Limitations. There are severe limitations that should be addressed to make theorem proving in Haskell a pleasant and usable technique. As mentioned earlier *the proofs are verbose*. There are a few cases where the proofs require domain specific knowledge. For example, to prove associativity of string matching `x ♢ (y ♢ z) = (x ♢ y) ♢ z` we need a theorem that performs case analysis on the relative length of the input field of `y` and the target string. Unlike this case split though, most proofs do not require domain specific knowledge and merely proceed by term rewriting and structural inductuction that should be automated via Coq-like [8] tactics or/and Dafny-like [54] heuristics. For example, synquid [76] could be used to automatically synthesize proof terms.

Currently, we suffer from two engineering limitations. First, all reflected function should exist in the same module, as reflection needs access to the function implementation that is unknown for imported functions. This is the reason why we need to use a user defined, instead of Haskell's built-in, list. In our implementation we used `CPP` as a current workaround of the one module restriction. Second, class methods cannot be currently reflected. Our current workaround is to define Haskell functions instead of class instances. For example (`append`, `nil`) and (`concatStr`, `emptyStr`) define the monoid methods of List and Refined String respectively.

Overall, we believe that the strengths outweigh the limitations which will be addressed in the near future, rendering Haskell a powerful theorem prover.

6.5 Conclusion

We made the first non-trivial use of (Liquid) Haskell as a proof assistant. We proved the parallelization of chunkable monoid morphisms to be correct and applied our parallelization technique to string matching, resulting in a formally verified parallel string matcher. Our proof uses refinement types to specify equivalence theorems, Haskell terms to express proofs, and

Liquid Haskell to check that the terms prove the theorems. Based on our 1839LoC sophisticated proof we conclude that Haskell can be successfully used as a theorem prover to prove arbitrary theorems about real Haskell code using SMT solvers to automate proofs over key theories like linear arithmetic and equality. However, Coq-like tactics or Dafny-like heurestics are required to ease the user from manual proof term generation.

Acknowledgments The material of this chapter have been submitted for publication as it may appear in ESOP 2017: Vazou, Niki; Polakow, Jeff. "Verified Parallel String Matching in Haskell".

Chapter 7

Related Work

LIQUID HASKELL combines ideas from four main lines of research areas. It is a refinement type checker (§ 7.1) that enjoys SMT-based (§ 7.2) automated type checking. Via Refinement Reflection we touch the expressiveness of fully dependently typed systems (§ 7.3), getting an automated and expressive verifier for Haskell programs (§ 7.4).

7.1 Refinement Types

Standard Refinement Types Refinement Types were introduced by Freeman and Pfenning [35], with refinements limited to restrictions on the structure of algebraic datatypes. Freeman and Pfenning carefully designed the refinement logic to ensure *decidable type inference* via the notion of predicate subtyping (PVS [81]). The goal of refinement types is to refine the type system of an existing, general purpose, target language so that it *rejects more programs* as ill typed, unlike dependent type systems, that aim to increase the expressiveness and alter the semantics of the language.

Applications of Refinement Types Xi and Pfenning implemented DML [106] a refinement type checker for ML where arrays are indexed by terms from Presburger arithmetic to statically eliminate array bound checking. Since then, refinement types have been implemented for various general purpose languages, including ML [7, 79], C [19, 80], Racket [49] and Scala [82] to prove various correctness properties ranging from safe memory accessing to correctness of security protocols. All the above systems operate under CBV semantics that implicitly assume that all free variables are bound to values. This assumption, that breaks under Haskell's lazy semantics,

turned out to be crucial for the soundness of refinement type checking. To restore soundness in LIQUID HASKELL we use a refinement type based termination checker to distinguish between provably terminating and potential diverging free variables.

Reconciliation between Expressiveness and Decidability Reluctant to give up decidable type checking, many systems have pushed the expressiveness of refinement types within decidable logics. Kawaguchi *et al.* [47] introduce *recursive* and *polymorphic* refinements for data structure properties increasing the expressiveness but also the complexity of the underlying refinement system. CATALYST [46] permits a form of higher order specifications where refinements are relations which may themselves be parameterized by other relations. However, to ensure decidable checking, CATALYST is limited to relations that can be specified as catamorphisms over inductive types, precluding for example, theories like arithmetic. In the same direction, Abstract and Bounded refinement types encode modular, higher order specifications using the decidable theory of uninterpreted functions. All the above systems only allow for "shallow" specifications, where the underlying solver can only reason about (decidable) abstractions of user defined functions and not the exact description of the function implementations of the functions. Refinement Reflection, on the other hand, reflects user defined function definitions into the logic, allowing for "deep" program specifications but requiring the user to manually provide cumbersome proof terms.

7.2 SMT-Based Verification

Even though refinement type systems use SMT solvers to achieve decidable program verification by highly constraining the expressiveness of specifications, SMT-based verification has been extensively used for program verification without the decidability constraint. In such verifiers the SMT solvers are used to decide validity of arbitrary (*i.e.* non strictly decidable) logics leading to expressive specifications but undecidable and unpredictable verification. Unpredictable verification, as described in [56], suffers from *the butterfly effect* as "a minor modification in one part of the program source causes changes in the outcome of the verification in other, unchanged and unrelated parts of the program". Here we present three SMT-based verifiers that have highly influenced the design decisions in LIQUID HASKELL and discuss the ways each one of them uses

to control the unpredictability of verification.

Sage [51] is a hybrid type checker. The specifications are expressed in the form of refinement types that allow predicates to be arbitrary terms of the language being typechecked. It uses the SMT solver Simplify [25] to statically discharge as many proof obligations as possible and defers the rest to runtime casts. LIQUID HASKELL is a subset of Sage that provably requires no runtime casts since predicates are carefully constrained to decidable logics. We used Knowles and Flanagan's formalism on denotational typing and correctness of Sage to formalize soundness and semantics of LIQUID HASKELL.

F* [88] is a refinement type checker that allows predicates to be arbitrary terms and aims to discharge all proof obligations via the SMT solver, leading to unpredictable verification. F* allows the user to control the SMT decision procedures by exposing to the user SMT tactics that can be used to direct verification in case of failure. Moreover, F* allows effectful computations (*e.g.* state, exceptions, divergence and IO) and combines refinement types with a sophisticated effect type system to reason about totality of programs. In (Liquid) Haskell all programs are pure, thus reasoning about effectful computations has already been taken care of by Haskell's basic (*i.e.* unrefined) type system that requires effectful computations to be wrapped inside monads. The only effects allowed in Haskell are exceptions and divergence that can be optionally tracked by LIQUID HASKELL's totality and termination checker respectively.

Dafny [54] is a prototype SMT-based verifier for imperative programs that allows arbitrarily expressive specifications in the form of pre- and post-conditions. Acknowledging the disadvantages of unpredictable verification, Dafny aims to give to the user control over the underlying SMT decision procedures via sophisticated trigger and fuel techniques. Dafny that verifies effectful, imperative code via pre- and post-conditions is quite different from LIQUID HASKELL that verifies the pure and functional Haskell via refinement types. Yet, the work of Leino and the rest of Dafny developers has been a great inspiration for existing and future work on challenges shared by both verifiers, including termination checking, coinduction [55], local calculations [57], and error reporting [53].

7.3 Dependent Type Systems

Dependent Types Systems like Coq [8], Agda [71], Idris [13] and Isabelle/HOL [73] express sophisticated theorems as types since they allow arbitrary terms to appear in the types. Constructive proofs of such theorems are just programs that are either manually written by the users or automatically generated via proof tactics and heuristics. However programs are not just proofs, thus, unlike LIQUID HASKELL, these verification oriented, dependent type systems fail to provide an environment for mainstream code development.

Expressiveness: Deep vs. Shallow Specifications Dependently typed languages permit *deep* specification and verification. To express and prove theorems, these systems represent and manipulate the exact descriptions of user-defined functions. For example, we can represent the specification that the list `append` function is associative and we can manipulate (unfold) its definition to write a small program that by reduction constructs a proof of the specification. On the other hand, standard refinement types, including LIQUID HASKELL without refinement reflection, restrict refinements to so-called *shallow* specifications that correspond to abstract interpretations of the behavior of functions within decidable logical domains. For example, refinements make it easy to specify that the list returned by the `append` function has size equal to the sum of those of its inputs but in the logic the exact definition of `append` is not known. Refinement Reflection reflects user defined functions into the logic allowing deep specifications. Verification still occurs using the abstract interpretations of the functions, but with refinement reflection, the abstraction of the function is exactly equal to its definition.

Proof Strategy: Type Level Computations vs. Abstract Interpretation Dependent type systems use type level computations to construct proofs by evaluation. On the other hand, in refinement type systems, safety proofs rely on validity of subtyping constraints that is checked externally by an SMT solver. That is, the type system is unable to perform any proof by evaluation, as the only information it has for each function is the abstraction that is described by its type. With refinement reflection, we fake type level computations: the Haskell, value level, proof terms provide all the required reduction steps that are then retrofitted as equality assertions to the SMT solver.

Automation: Tactics vs. SMT solvers Dependent type checking requires explicit proof terms to be provided by the users. To help automate proof term generation, both built-in and user-provided tactics and heuristics are used to attempt to discharge proof obligations; however, the user is ultimately responsible for manually proving any obligations which the tactics are unable to discharge. On the other hand, refinement type checking does not require explicit proof terms. Verification proceeds by checking validity of subtyping constraints which reduces to implication checking that is in turn decided using the power of SMT solvers. Many times the SMT solver fails to prove validity of a subtyping constraint because the environment is too weak. In such cases the user can strengthen the environment by instantiating axioms via function calls. For example, the proof that `() :: {v:() | fib 1 = 1}` is valid under an environment that invokes `fib` on `1`. That is, to prove deep specifications we fake type level computations via value level proof terms, but ultimately, we check validity using the power of SMTs which drastically simplifies proofs over key theories like linear arithmetic and equality. In the future, we plan to investigate how to simplify such proof terms by adapting the tactics and heuristics of dependently typed systems into LIQUID HASKELL.

System Features: Theorem Prover vs. Legacy Programming Languages Dependent type systems are proof oriented systems, lacking features required for a general purpose language, like diverging and effectful programs. Various systems extend theorem provers to support effectful programs, for example Zombie [15, 85] and F* [88] allow dependent types to coexist with divergent and effectful programs. Still, these systems are verification oriented and lack the optimized libraries that come from the mainstream developers of a general purpose programming language. On the other hand, LIQUID HASKELL retrofits verification in Haskell, a legacy programming language with a long-standing developer community. With LIQUID HASKELL, Haskell programmers can as use their favorite language for general purpose programming, and also prove specifications without the need to use an external, verification specific, theorem prover.

7.4 Haskell Verifiers

LIQUID HASKELL belongs into the ongoing research of Haskell code verification that is exploring techniques to verify properties about Haskell programs that the current type system cannot specify. There are two main directions in this line of research. Some groups are building external verifiers that analyze well typed Haskell programs, while others are enriching the expressiveness of the Haskell's type system.

Domain Specific Haskell Verifiers Various external Haskell analyzers have been proposed to check correctness properties of Haskell code that is not expressible by Haskell's type system. Catch [65] is a fully automated tool that tracks incomplete patterns, like our totality analyzer. AProVE [36] implements a powerful, fully-automatic termination analysis for Haskell based on term-rewriting, like our termination analyzer. HERMIT [31] proves equalities by rewriting the GHC core language, guided by user specified scripts, like our equality reasoning performed via Refinement Reflection. All the above verifiers allow for a domain specific analysis, precluding LIQUID HASKELL's generalized functional correctness specifications, encoded via refinement typing.

Static Contract Checking A generalized correctness analysis in Haskell is feasible via Haskell's static contract checking [107] that encodes arbitrary contracts in the form of refinement types and checks them using symbolic execution to unroll procedures upto some fixed depth. Similarly, Zeno [86] is an automatic Haskell prover that combines unrolling with heuristics for rewriting and proof-search. Finally, the Halo [103] contract checker encodes Haskell programs into first-order logic by directly modeling the code's denotational semantics, again, requiring heuristics for instantiating axioms describing functions' behavior. All the above general purpose verifiers allow specification of arbitrarily expressive contracts rendering verification undecidable and thus impractical.

Dependent Types in Haskell Haskell itself is a dependently-typed language [27], as type level computation is allowed via Type Families [63], Singleton Types[29], Generalized Algebraic Datatypes (GADTs) [74, 83], type-level functions [16], and explicit type applications [30]. In

this line of work [28] Eisenberg *et al.* aim to allow fully dependent programming within Haskell, by making "type-level programming ... at least as expressive as term-level programming". Our approach differs in two significant ways. First, while enriching expressiveness of the types allows Haskell's type system to accept more programs, we aim not to alter semantics of Haskell programs, but by refining the checks performed by the type system to reject more programs as ill typed. As a consequence, refinements are completely erased at run-time. As an advantage (resp. disadvantage), refinements cannot degrade (resp. optimize) the performance of programs. Second, dependent Haskell follows the classic dependent type verification by type level evaluation approach that turns out to be quite painful [59]. On the other hand, LIQUID HASKELL enjoys SMT-aided verification, which drastically simplifies proofs over key theories like linear arithmetic and equality. Despite these differences, these two approaches target the same problem of lifting value level terms into Haskell's type system. In the future, we hope to unify these two techniques and allow a uniform interface for lifting values inside the type specifications to create a dependent Haskell that enjoys both SMT-based automation of verification and type driven runtime optimizations.

Chapter 8

Conclusion

We presented LIQUID HASKELL, an automatic, sound, and expressive verifier for Haskell code. We started (Chapter 1) by porting standard refinement types to Haskell to verify more than 10K lines of popular Haskell libraries. Then (Chapter 2), we observed that Haskell's lazy semantics render standard refinement type checking unsound and restored soundness via a refinement type based termination checker. Next, we presented Abstract (Chapter 3) and Bounded (Chapter 4) Refinement Types, that use uninterpreted functions to *abstract* and *bound* over the refinements of the types. We used both these techniques to encode higher order, modular specifications while preserving SMT based decidable and predictable type checking. Finally, we presented Refinement Reflection (Chapter 5) a technique that reflects terminating, user defined, Haskell functions into the logic, turning (Liquid) Haskell into an arbitrarily expressive theorem prover. We used LIQUID HASKELL to prove correctness of sophisticated properties ranging from safe memory indexing to code equivalence over parallelization (Chapter 6)

In short, we described how to turn Haskell into a theorem prover that enjoys both the SMT-based automatic and predictable type checking of refinement types and the optimized libraries and parallel runtimes of the mature, general purpose language Haskell.

In the future we plan to use LIQUID HASKELL as an interactive environment that, using techniques of code synthesis and error diagnosis, will integrate formal verification into the mainstream development process to aid, rather than complicate, code development.

Bibliography

[1] N. Amin, K. R. M. Leino, and T. Rompf. Computing with an SMT Solver. In *TAP*, 2014.

[2] L. Augustsson. Cayenne - A language with dependent types. In *ICFP*, 1998.

[3] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, 2011.

[4] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0, 2010.

[5] G. Barthe and O. Pons. Type isomorphisms and proof reuse in dependent type theory. In *FoSSaCS*. Springer, 2001.

[6] J. F. Belo, M. Greenberg, A. Igarashi, and B. C. Pierce. Polymorphic contracts. In *ESOP*, 2011.

[7] J. Bengtson, K. Bhargavan, C. Fournet, A. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF*, 2008.

[8] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions"*. Springer Verlag", 2004".

[9] G. E. Blelloch. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Pub, 1993.

[10] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *HotPar*, 2009.

[11] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Rossum, S. Schulz, and R. Sebastiani. MathSAT: Tight integration of SAT and mathematical decision procedures. *J. Autom. Reason.*, 2005.

[12] A. Bradley and Z. Manna. *The Calculus of Computation: Decision Procedures With Application To Verification*. Springer-Verlag, 2007.

[13] E. Brady. Idris: general purpose programming with dependent types. In *PLPV*, 2013.

[14] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *OOPSLA*, 2010.

[15] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. In *POPL*, 2014.

[16] M. T. Chakravarty, G. Keller, and S. L. Peyton-Jones. Associated type synonyms. In *ICFP*, 2005.

[17] M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problem. In *Parco*, 1993.

[18] S. Collange, D. Defour, S. Graillat, and R. Iakymchuk. Full-Speed Deterministic Bit-Accurate Parallel Floating-Point Summation on Multi- and Many-Core Architectures. https://hal.archives-ouvertes.fr/hal-00949355, 2014.

[19] J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *ESOP*, 2007.

[20] R. L. Constable and S. F. Smith. Partial objects in constructive type theory. In *LICS*, 1987.

[21] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs. In *POPL*, 1977.

[22] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, 2011.

[23] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. In *ICFP*, 2007.

[24] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. *TACAS*, 2008.

[25] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 2005.

[26] J. Dunfield. Refined typechecking with Stardust. In *PLPV*, 2007.

[27] R. A. Eisenberg. Dependent types in haskell: Theory and practice. *CoRR*, 2016.

[28] R. A. Eisenberg and J. Stolarek. Promoting functions to type families in Haskell. In *Haskell*, 2014.

[29] R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *Haskell*, 2012.

[30] R. A. Eisenberg, S. Weirich, and H. G. Ahmed. Visible type application. In *ESOP*, 2016.

[31] A. Farmer, N. Sculthorpe, and A. Gill. Reasoning with the HERMIT: Tool support for equational reasoning on GHC Core programs. In *Haskell*, 2015.

[32] J. Filliâtre. Proof of imperative programs in type theory. In *TYPES*, 1998.

[33] C. Flanagan, R. Joshi, and K. R. M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 2001.

[34] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *PODC*, 2004.

[35] T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, 1991.

[36] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for Haskell by term rewriting. *TPLS*, 2011.

[37] D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operationsd. In *POPL*, 2005.

[38] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 1993.

[39] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*, 1971.

[40] G. P. Huet. The Zipper. *J. Funct. Program.*, 1997.

[41] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL*, 1996.

[42] G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[43] J. JáJá. *Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, 1992.

[44] R. Jhala and K. L. McMillan. Array abstractions from proofs. In *CAV*, 2007.

[45] S. Kahrs. Red-black trees with types. *J. Funct. Program.*, 2001.

[46] G. Kaki and S. Jagannathan. A relational framework for higher-order shape analysis. In *ICFP*, 2014.

[47] M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI*, 2009.

[48] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *ICFP*, 2010.

[49] A. M. Kent, D. Kempe, and S. Tobin-Hochstadt. Occurrence typing modulo theories. In *PLDI*, 2016.

[50] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell*, 2004.

[51] K. Knowles and C. Flanagan. Hybrid type checking. *ACM TOPLAS*, 2010.

[52] L. Kuper, A. Turon, N. R. Krishnaswami, and R. R. Newton. Freeze after writing: quasi-deterministic parallel programming with lvars. In *POPL*, 2014.

[53] C. Le Goues, K. R. M. Leino, and M. Moskal. The boogie verification debugger (tool paper). In *Software Engineering and Formal Methods*, 2011.

[54] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010.

[55] K. R. M. Leino and M. Moskal. Co-induction simply - automatic co-inductive proofs in a program verifier. In *Formal Methods*, 2014.

[56] K. R. M. Leino and C. Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In *CAV*, 2016.

[57] K. R. M. Leino and N. Polikarpova. Verified calculations. In *VSTTE*, 2016.

[58] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL 06*, 2006.

[59] S. Lindley and C. McBride. Hasochism: the pleasure and pain of dependently typed Haskell programming. In *Haskell*, 2013.

[60] F. Loulergue, W. Bousdira, and J. Tesson. Calculating Parallel Programs in Coq using List Homomorphisms. In *International Journal of Parallel Programming*, 2016.

[61] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for haskell. In *Haskell*, 2010.

[62] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Haskell*, 2011.

[63] C. McBride. Faking it: Simulating dependent types in Haskell. *J. Funct. Program.*, 2002.

[64] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional GPU programs. In *ICFP*, 2013.

[65] N. Mitchell and C. Runciman. Not all patterns, but enough - an automatic verifier for partial but sufficient pattern matching. In *Haskell*, 2008.

[66] S. Moore, C. Dimoulas, D. King, and S. Chong. SHILL: A secure shell scripting language. In *OSDI*, 2014.

[67] S.-c. Mu, H.-s. Ko, and P. Jansson. Algebra of Programming in Agda: Dependent Types for Relational Program Derivation. *J. Funct. Program.*, 2009.

[68] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, 2008.

[69] G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.

[70] T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In *CSL*, 2002.

[71] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers, 2007.

[72] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *IFIP TCS*, 2004.

[73] L. C. Paulson. Isabelle A Generic Theorem prover. *Lecture Notes in Computer Science*, 1994.

[74] S. L. Peyton-Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP*, 2006.

[75] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[76] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, 2016.

[77] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *25th ACM National Conference*, 1972.

[78] S. R. D. Rocca and L. Paolini. *The Parametric Lambda Calculus, A Metamodel for Computation*. Springer Science and Business Media, 2004.

[79] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

[80] P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL*, 2010.

[81] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in pvs. *IEEE TSE*, 1998.

[82] G. S. Schmid and V. Kuncak. SMT-based Checking of Predicate-Qualified Types for Scala. In *Scala*, 2016.

[83] T. Schrijvers, S. L. Peyton-Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *ICFP*, 2009.

[84] T. Sheard. Type-level computation using narrowing in omega. In *PLPV*, 2006.

[85] V. Sjöberg and S. Weirich. Programming up to congruence. *POPL*, 2015.

[86] W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. In *TACAS*, 2012.

[87] M. Sulzmann, M. M. T. Chakravarty, S. L. Peyton-Jones, and K. Donnelly. System F with type equality coercions. In *TLDI*, 2007.

[88] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *POPL*, 2016.

[89] W. Swierstra. Xmonad in Coq (experience report): Programming a window manager in a proof assistant. In *Haskell*, 2012.

[90] T. L. H. Team. github.com/ucsd-progsys/liquidhaskell/tree/master/benchmarks/icfp15.

[91] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *ICFP*, 2010.

[92] G. Tourlakis. Ackermanns Function. http://www.cs.yorku.ca/~gt/papers/Ackermann-function.pdf, 2008.

[93] A. M. Turing. On computable numbers, with an application to the eintscheidungsproblem. In *LMS*, 1936.

[94] H. Unno, T. Terauchi, and N. Kobayashi. Relatively complete verification of higher-order functional programs. In *POPL*, 2013.

[95] N. Vazou, A. Bakst, and R. Jhala. Technical report: Bounded Refinement Types, 2015. https://github.com/nikivazou/thesis/blob/master/techreps/icfp15.pdf.

[96] N. Vazou, V. Choudhury, R. G. Scott, R. Jhala, and R. R. Newton. Technical report: Refinement Reflection: Parallel legacy languages as theorem provers, 2016. https://github.com/nikivazou/thesis/blob/master/techreps/pldi16.pdf.

[97] N. Vazou and J. Polakow. Code for verified string indexing, 2016. https://github.com/nikivazou/verified_string_matching.

[98] N. Vazou, P. Rondon, and R. Jhala. Abstract refinement types. In *ESOP*, 2013.

[99] N. Vazou, E. L. Seidel, and R. Jhala. Liquidhaskell: Experience with refinement types in the real world. In *Haskell Symposium*, 2014.

[100] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement Types for Haskell. In *ICFP*, 2014.

[101] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Technical report: Refinement Types for Haskell, 2014. https://github.com/nikivazou/thesis/blob/master/techreps/icfp14.pdf.

[102] P. Vekris, B. Cosman, and R. Jhala. Refinement types for typescript. In *PLDI*, 2016.

[103] D. Vytiniotis, S. Peyton-Jones, K. Claessen, and D. Rosén. Halo: haskell to logic through denotational semantics. In *POPL*, 2013.

[104] G. Wiki. GHC optimisations. https://wiki.haskell.org/GHC_optimisations.

[105] H. Xi. Dependent types for program termination verification. In *LICS*, 2001.

[106] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, 1998.

[107] D. N. Xu, S. L. Peyton-Jones, and K. Claessen. Static contract checking for haskell. In *POPL*, 2009.