UNIVERSITY OF CALIFORNIA SAN DIEGO

Mechanizing Refinement Types

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

in

Computer Science

by

Michael H. Borkowski

Committee in charge:

Professor Ranjit Jhala, Chair
Professor Samuel R. Buss
Professor Cormac Flanagan
Professor Nadia Polikarpova
Professor Victor Vianu

2024

The dissertation of Michael H. Borkowski is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2024

DEDICATION

To Kiyoshi, Daikichi, and Ziggy

TABLE OF CONTENTS

Dissertation Approval Page iii

Dedication iv

Table of Contents v

List of Figures vii

List of Tables viii

Acknowledgements ix

Vita xi

Abstract of the Dissertation xii

Chapter 1 Introduction 1
1.1 Outline 2
1.2 Related Work 4

Chapter 2 Refinement Types 8
2.1 The goal of Refinement Types 8
2.2 The Essence of Refinement Types 11
2.3 The Design of Refinement Types 11
2.3.1 Semantic Subtyping 12
2.3.2 Decidability 13
2.3.3 Polymorphism 15
2.4 The Soundness of Refinement Types 16

Chapter 3 The Languages λ_F and λ_{RF} 21
3.1 Syntax 21
3.2 Dynamic Semantics 23
3.3 Static Semantics 26
3.3.1 Well-formedness 26
3.3.2 Typing 28
3.3.3 Subtyping 32
3.3.4 Implication 33

Chapter 4 λ_F Soundness 38
4.1 Static Semantics 39
4.2 Metatheory for λ_F 39
4.2.1 Progress 41

4.2.2 Preservation 45

Chapter 5 Soundness of λ_{RF} 52
5.1 Denotational Soundness 53
5.2 Type Safety 54
5.3 Inversion of Typing Judgments 55
5.4 Substitution Lemma 56
5.5 Narrowing 57

Chapter 6 LIQUIDHASKELL & Refined Data Propositions 59
6.1 LIQUIDHASKELL 59
6.2 Refined Data Propositions 61

Chapter 7 Implementation and Mechanization 65
7.1 LIQUIDHASKELL Mechanization 65
7.1.1 Quantitative Results 66
7.2 COQ Mechanization 68

Chapter 8 Comparison of Proof Assistants 69
8.1 Proving Theorems in LIQUIDHASKELL 69
8.1.1 SMT Solvers, Arithmetic, and Set Theory 69
8.1.2 Co-finite Quantification 71
8.1.3 Inductive Proofs as Recursive Functions 72
8.2 COQ *vs.* LIQUIDHASKELL 74

Chapter 9 Lists: The Language λ_{RFD} 79
9.1 Syntax and Semantics 79
9.1.1 Subtyping 84
9.1.2 Denotational Semantics 86
9.2 Metatheory of Lists 86
9.3 Implementation 87

Chapter 10 Conclusions & Future Work 89

LIST OF FIGURES

Figure 2.1: Functional Arrays with refinement types that ensure safe indexing. 9
Figure 2.2: Dependencies of Typing Judgements in Refinement Types. (Dashed lines do not exist in our formalism.) 17

Figure 3.1: Syntax of Primitives, Values, and Expressions. 22
Figure 3.2: Syntax of Types. The gray boxes are the extensions to λ_F needed by λ_{RF}. We use τ for λ_F-only types. 22
Figure 3.3: The small-step semantics. 24
Figure 3.4: Type substitution and refinement strengthening. 25
Figure 3.5: Well-formedness of types and environments. The rules for λ_F exclude the gray boxes. 28
Figure 3.6: Typing rules. The judgment $\Gamma \vdash_F e : \tau$ is defined by excluding the gray boxes. 29
Figure 3.7: Subtyping Rules. 33
Figure 3.8: Denotations of Types and Environments. 36

Figure 4.1: Well-formedness of λ_F types. 39
Figure 4.2: Unrefined typing rules. 40

Figure 5.1: Dependencies in the metatheory. We write "var" and "tv" to resp. abbreviate term and type variables. 53

Figure 8.1: Encoding of Co-finitely Quantified Rules. 72

Figure 9.1: Syntax of Primitives, Values, and Expressions. 80
Figure 9.2: Syntax of Types. The gray boxes are the extensions to λ_F needed by λ_{RFD}. 80
Figure 9.3: The small-step semantics for λ_{RFD}. 81
Figure 9.4: Well-formedness of λ_{RFD} types. The rules for λ_F exclude the gray boxes. . 82
Figure 9.5: Typing rules. The judgment $\Gamma \vdash_F e : \tau$ is extended by excluding the gray boxes. 83
Figure 9.6: Subtyping Rules. 84
Figure 9.7: Denotations of Types and Environments. 86

LIST OF TABLES

Table 7.1: Quantitative mechanization details. We split each development into sets of modules pertaining to regions of Figure 5.1 and for each we count lines of specification (definitions, lemma statements) and of proof. 67

Table 9.1: Comparative mechanization details for λ_{RF} versus λ_{RFD}. 88

ACKNOWLEDGEMENTS

First, I would like to thank Ranjit Jhala for all of his support as my advisor. I am greatly indebted to Ranjit for taking me on as his student when I was a complete beginner at type theory and software verification research. He provided the original motivation for my work in his 2019 graduate class on LIQUIDHASKELL, and I've been hooked on theorem proving ever since. I appreciate Ranjit's insights and feedback during our meetings and, most of all, his continuing confidence in my work throughout four conference rejections motivated me to keep improving and adding to our work.

I would also like to thank my collaborator and coauthor Niki Vazou for all of her patient help, support, and ideas. I couldn't have done this research without her support either! I want to thank each of the members of my committee, Nadia Polikarpova, Victor Vianu, Sam Buss, and Cormac Flanagan for their support through this process and for the opportunity to TA for some of their classes as well.

I want to thank my wife Ashley and our sons Kiyoshi, Daikichi, and Zygmunt for their patience and support for the many hours that I spent away from them working on the mechanizations and on this dissertation.

I want to thank my fellow PL students for many helpful conversations, and especially Saketh Kasibatla, Kyle Thompson, and Cole Kurashige for helpful conversations about COQ and theorem proving. I also thank James Parker for a helpful discussion about data propositions and the anonymous reviewers across five conferences for their useful comments and suggestions. I owe a debt of gratitude to Joe Politz and Sorin Lerner for detailed comments and feedback on an early version of my POPL 24 talk.

Work adapted in this dissertation

Chapters 1-3, 5-8, and the conclusion are adapted from "Mechanizing Refinement Types" in the proceedings of the 51^{st} ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2024), by Michael Borkowski, Niki Vazou, and Ranjit Jhala.

Chapter 4 is adapted from unpublished material that was originally prepared for the same "Mechanizing Refinement Types" by Michael Borkowski, Niki Vazou, and Ranjit Jhala but did not appear in the final published version.

Chapter 9 describes unpublished work done in collaboration with Ranjit Jhala.

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

VITA

2016	B. A. in Computer Science *magna cum laude*, Amherst College
2019	M. S. in Computer Science, University of California San Diego
2024	Ph. D. in Computer Science, University of California San Diego

PUBLICATIONS

M. H. Borkowski, N. Vazou, and R. Jhala, "Mechanizing Refinement Types", *POPL*, 2024.

ABSTRACT OF THE DISSERTATION

Mechanizing Refinement Types

by

Michael H. Borkowski

Doctor of Philosophy in Computer Science

University of California San Diego, 2024

Professor Ranjit Jhala, Chair

Practical checkers based on refinement types use the combination of implicit semantic subtyping and parametric polymorphism to simplify the specification and automate the verification of sophisticated properties of programs. However, a formal metatheoretic accounting of the *soundness* of refinement type systems using this combination has proved elusive. We present λ_{RF}, a core refinement calculus that combines semantic subtyping and parametric polymorphism. We develop a metatheory for this calculus and prove soundness of the type system. We give two mechanizations of our metatheory. First, we introduce *data propositions*, a novel feature that enables encoding derivation trees for inductively defined judgments as refined data types, and use them to show that LIQUIDHASKELL's refinement types can be used *for* mechanization. Second,

we mechanize our results in COQ, which comes with stronger soundness guarantees than LIQUIDHASKELL, thereby laying the foundations for mechanizing the metatheory *of* LIQUIDHASKELL. Finally, we present an extension λ_{RFD}, which adds lists and a length measure. We extend the metatheory to prove the soundness of the extended type system and give another mechanization in COQ.

Chapter 1

Introduction

Refinements constrain types with logical predicates to specify new concepts. For example, the refinement type $\texttt{Pos} \doteq \texttt{Int}\{\texttt{v} : 0 < v\}$ describes *positive* integers and $\texttt{Nat} \doteq \texttt{Int}\{\texttt{v} : 0 \leq v\}$ specifies natural numbers. Refinement types have been successfully used to specify various properties like secrecy [17], resource usage [25], or information flow [28] that can then be verified in programs developed in various programming languages like Haskell [51], Scala [22], and Racket [24].

The success of refinement types relies on the combination of two essential features. First, *implicit* semantic subtyping uses semantic (SMT-based) reasoning to automatically convert the types of expressions without hassling the programmer for explicit type casts. For example, consider a positive expression $e : \texttt{Pos}$ and a function expecting natural numbers $f : \texttt{Nat} \to \texttt{Int}$. To type check the application $f\ e$, the refinement type system will implicitly convert the type of e from `Pos` to `Nat`, because $0 < v \Rightarrow 0 \leq v$ holds semantically. Importantly, refinement types propagate semantic subtyping inside type constructors to, for example, treat function arguments in a contravariant manner. Second, *parametric polymorphism* allows the propagation of the refined types through polymorphic function interfaces, without the need for extra reasoning. As a trivial example, once we have established that e is positive, parametric polymorphism should let us

conclude that $g\ e$: `Pos` if, for example, g is the identity function $g : a \to a$. As a more interesting example, in § 2.1 we combine semantic subtyping and polymorphism to verify a safe-indexing array of prime numbers.

The engineering of practical refinement type checkers has galloped far ahead of the development of their metatheoretical foundations. In fact, semantic subtyping is very tricky as it is mutually defined with typing, leading to metatheoretic proofs with circular dependencies (Figure 2.2). Unsurprisingly, the addition of polymorphism poses further challenges. As Sekiyama et al. [44] observe, a naïve definition of type instantiation can lose potentially contradicting refinements leading to unsoundness. Existing formalizations of refinement types drop semantic subtyping [44, 22] or polymorphism [15, 48], or have problematic metatheory [4].

1.1 Outline

In this dissertation we formalize λ_{RF}, a core calculus with a refinement type system that combines semantic subtyping with polymorphism, via five concrete contributions. But first, we begin in Chapter 2 with an overview of refinement types, giving examples of their applications and discussing their essential features. We conclude the chapter in § 2.4 by outlining the challenges we encountered in attempting to prove the soundness of λ_{RF} and how we addressed each of them.

1. Reconciliation In Chapter 3 we introduce our first contribution: a language that combines refinements and polymorphism in a way that ensures the metatheory remains sound without sacrificing the expressiveness needed for practical verification. To this end, λ_{RF} introduces a kind system that distinguishes the type variables that can be soundly refined (without the risk of losing refinements at instantiation) from the rest, which are then left unrefined. In addition, our design includes a form of existential typing [26] which is essential to *synthesize* the types – in the sense of bidirectional typing – for applications and let-binders in a compositional manner (§ 2.3.2, § 3.3).

2. Foundation Our second contribution, described in Chapter 5, is to establish the foundations of λ_{RF} by proving soundness, which says that well-typed expressions cannot get stuck and belong to the denotation of their type (§ 5.1, § 5.2). The combination of semantic subtyping, polymorphism, and existentials makes the soundness proof challenging with circular dependencies that do not arise in standard (unrefined) calculi. The mechanization was simplified by the use of two essential ingredients. First, we use an unrefined *base* language λ_F, a classic System F [38], in rules where refinements are not required, cutting two potential circularities in the static judgments (Figure 2.2). Second, we define an *implication interface* that abstractly specifies the properties of implication required to prove type soundness, and show how this interface can be implemented via denotational semantics (§ 3.3.4).

3. Reification Our third contribution, presented in Chapter 6, is to introduce *data propositions*, a novel feature in LIQUIDHASKELL that enables the encoding of derivation trees for inductively defined judgments as refined data types, by first reifying the propositions and evidence as plain Haskell data, and then using refinements to connect the two. Hence, data propositions let us write plain Haskell functions over refined data to provide explicit, constructive proofs (§ 6.2). Without data propositions reasoning about potentially non-terminating computations was not possible in LIQUIDHASKELL, thereby precluding even simple metatheoretic developments such as the soundness of λ_F let alone λ_{RF}.

4. Mechanization Chapter 7 describes another contribution: we mechanized the metatheory of λ_{RF} *twice*: using LIQUIDHASKELL and COQ. We formalized λ_{RF} in LIQUIDHASKELL (§ 7.1) to evaluate the feasibility of such substantial metatheoretical formalizations. Our proof is non-trivial, requiring 9,400 lines of code, 30 minutes to verify, and various modifications in the internals of LIQUIDHASKELL. We translated the same proof to COQ (§ 7.2) to compare the two alternatives. Certain definitions, concretely the type denotations, not admissible by LIQUIDHASKELL's positivity checker, were possible to define in COQ using Equations [45]. Further, the COQ development is much faster (about 60 seconds to verify), but it is also more dif-

ficult to manipulate various partial and mutual recursive definitions of the formalization. Finally, COQ comes with stronger foundational soundness guarantees than LIQUIDHASKELL. While the metatheory of COQ is well studied, λ_{RF} lays the foundation for the mechanized metatheory of LIQUIDHASKELL.

5. Data Types Our next contribution, described in Chapter 9, is to add data types to λ_{RF} in the form of lists that are equipped with a length measure. For instance, the programmer can write a safe polymorphic tail function that requires its input to be a non-empty list (length at least one). The addition of lists to our language, which we denote by λ_{RFD}, adds new complexity to our metatheory. A case switch operator enables path-sensitive reasoning when destructing lists and leads to challenging new cases for our soundness theorems.

1.2 Related Work

We discuss the most closely related work on the metatheory of unrefined and refined type systems.

Hybrid & Contract Systems Flanagan [15] formalizes on paper a monomorphic lambda calculus with refinement types that differs from our λ_{RF} in two ways. First, in Flanagan [15]'s type checking is hybrid: the developed system is undecidable and inserts runtime casts when subtyping cannot be statically decided. Second, the original system lacks polymorphism. Sekiyama et al. [44] extended hybrid types with polymorphism, but unlike λ_{RF}, their system does not support semantic subtyping. For example, consider a divide by zero-error. The refined types for `div` and 0 could be given by div :: $\texttt{Int} \to \texttt{Int}\{n : n \neq 0\} \to \texttt{Int}$ and $0 :: \texttt{Int}\{n : n = 0\}$. This system will compile `div 1 0` by inserting a cast on 0: $\langle \texttt{Int}\{n : n = 0\} \Rightarrow \texttt{Int}\{n : n \neq 0\} \rangle$, causing a definite runtime failure that could have easily been prevented statically. Having removed semantic subtyping, the metatheory of Sekiyama et al. [44] is highly simplified. Static refinement type systems (as summarized by Jhala and Vazou [23]) usually restrict the definition of predicates to

quantifier-free first-order formulae that can be *decided* by SMT solvers. This restriction is not preserved by evaluation that can substitute variables with any value, thus allowing expressions that cannot be encoded in decidable logics, like lambdas, to seep into the predicates of types. In contrast, we allow predicates to be any language term (including lambdas) to prove soundness via preservation and progress: our meta-theoretical results trivially apply to systems that, for efficiency of implementation, restrict their source languages. Finally, none of the above systems (hybrid, contracts or static refinement types) come with a machine checked soundness proof.

Semantic Subtyping Semantic subtyping is not a unique feature of refinement types. For example, Frisch et al. [18] use the set theoretic models of types to decide subtyping. Castagna and Frisch [8] present an algorithm that decides semantic subtyping for a core calculus with functional types. Like λ_{RF}, Castagna and Frisch [8] introduce a denotational interpretation of types to break the circularity between the typing and subtyping relations. Unlike λ_{RF}, their system does not have polymorphism and, crucially, has no notion of dependency (no refinement type-style binder of arguments). Moreover, their subtyping algorithm is different from our refinement based algorithm: it is neither type directed nor efficient (*i.e.* it requires backtracking), and cannot be automated by an external SMT solver.

Mechanizations of Refinement Types Lehmann and Tanter [27]'s COQ formalization of a monomorphic, refined calculus differs from λ_{RF} in two ways. First, their axiomatized implication, which is similar to our implication interface, allows them to restrict the language of refinements to decidable logics but provides no formal connection between subtyping and evaluation. Instead, we also provide the denotational implementation of the implication interface, thus establish denotation soundness. Second, λ_{RF} includes polymorphism, existentials, and selfification which are critical for context-sensitive refinement typing, but make the metatheory more challenging. Hamza et al. [22] present System FR, a polymorphic, refined language with a mechanized metatheory of about 30K lines of COQ. Compared to our system, their notion of subtyping is not semantic, but relies on a reducibility relation. For example, even though System

FR will deduce that `Pos` is a subtype of `Int`, it will fail to derive that `Int` $\rightarrow$ `Pos` is subtype of `Pos` $\rightarrow$ `Int` as reduction-based subtyping cannot reason about contra-variance. Because of this more restrictive notion of subtyping, their mechanization requires neither the indirection of denotational soundness nor an implication proving oracle. Further, System FR's support for polymorphism is limited in that it disallows refinements on type variables, thereby precluding many practically useful specifications. Recently, Chen [10] formalized a refinement type system as an embedding of refinement types in Agda. This system is verified in a few thousand lines of Agda. This formalism differs significantly from ours in that as an embedding it is built on top of a rich theorem prover and cannot be used to refine some existing programming language. Further, it does not support higher-order functions, polymorphism, semantic subtyping, neither be automated by an external solver since soundness reduces to Agda's soundness. Finally, Ghalayini and Krishnaswami [19] mechanize refinement types with explicit proof terms in 15K lines of LEAN code. They use a categorical, denotational semantics soundness statement, but their calculus by design supports neither semantic subtyping nor polymorphism.

***Metatheory in* LIQUIDHASKELL** `LWeb` [36] also used LIQUIDHASKELL to prove metatheory, the non-interference of λ_{LWeb}, a core calculus that extends the LIO formalism with database access. The `LWeb` proof did not use refined data propositions, which were not present at development time, and thus it has two major weaknesses compared to our present development. First, `LWeb` *assumes* termination of λ_{LWeb}'s evaluation function; without refined data propositions metatheory can be developed only over terminating functions. This was not a critical limitation since non-interference was only proved for terminating programs. However, in our proof the requirement that evaluation of λ_{RF} terminates would be too strict. In our encoding with refined data propositions such an assumption was not required. Second, the `LWeb` development is not constructive: the structure of an assumed evaluation tree is logically inspected instead of the more natural case splitting permitted only with refined data propositions. This constructive way to develop metatheories is more compact (*e.g.* there is no need to logically inspect derivation

trees) and akin to the standard meta-theoretic developments of constructive tools like COQ and ISABELLE.

Acknowledgements for Chapter 1

This chapter is adapted from "Mechanizing Refinement Types" in the proceedings of the 51^{st} ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2024), by Michael Borkowski, Niki Vazou, and Ranjit Jhala. The dissertation author was the primary investigator and author of this material.

Chapter 2

Refinement Types

We start with an informal overview of the usefulness of refinement types and of our refined core calculus λ_{RF}, which we later present formally (Chapter 3) and prove sound (Chapter 5). Concretely, we present the goals of refinement types (§ 2.1) and how they are achieved via three essential features: semantic subtyping, existential types, and polymorphism (§ 2.2). We explain how the typing judgements are designed to accommodate these features (§ 2.3) and how we addressed the challenges these features impose in the mechanization of the soundness proof (§ 2.4). Our examples here are presented with the syntax of LIQUIDHASKELL, but can be encoded in λ_{RF}.

2.1 The goal of Refinement Types

Refinement types refine the types of an existing programming language with logical predicates to define abstractions not expressible by the underlying type system, which can then be used for static (1) error prevention and (2) functional correctness.

Error Prevention Figure 2.1 presents the interface of a fixed size array that is encoded in the core calculus λ_{RF} as a function. The function `new n x` returns an array that contains `x` when indexed with an integer between `0` and `n` and otherwise throws an "out of bounds" error.

```
type ArrayN a N = {i:Nat | i < N} → a

new :: n:Nat → a → ArrayN a n
new n x = \i → if 0 ≤ i && i < n then x else error "Out of Bounds"

set :: n:Nat → i:{Nat | i < n} → a → ArrayN a n → ArrayN a n
set n i x a = \j → if i == j then x else a j

get :: n:Nat → i:{Nat | i < n} → ArrayN a n → a
get n i a = a i
```

Figure 2.1: Functional Arrays with refinement types that ensure safe indexing.

To statically ensure that this error will never occur, `new` returns the refined array `ArrayN a n`, *i.e.* a function whose domain is restricted to integers less than `n`. The `set` and `get` operators manipulate the refined arrays on the index `i:{Nat | i < n}`, *i.e.* refined to be in-bounds of the array. With this refined interface, out-of-bounds indexing is statically ruled out:

```
array10 :: ArrayN Int 10

array10 = new 10 0


good = get 10 4 array10 -- OK

bad = get 10 42 array10 -- Refinement Type Error
```

Functional Correctness Refinement types are also used to ensure that the program has the intended behavior. To achieve this, we use uninterpreted functions to *specify* behaviors and rely on the type system to propagate them. For example, below using the uninterpreted function `isPrime` we *specify* that some integers are primes, as denoted by the *uninterpreted* predicate `isPrime`.

```
measure isPrime :: Int → Bool

type Prime = {v:Int | isPrime v}
```

Refinement types are not ideally suited to verifying properties like primality checking, which requires reasoning beyond SMT decidable fragments. However, *assuming* that a function establishes

primality, refinements can be used to easily track and propagate the invariant:

```
assume checkPrime :: x:Int → {v:Bool | v ⇔isPrime x}


nextPrime :: Nat → Prime
nextPrime x = if checkPrime x then x else nextPrime (x+1)
```

The path-sensitivity of refinement types (Rule T-IF of Figure 3.6) ensures that the function `nextPrime` returns only values that pass the primality check.

Note on recursion Our core calculus does not explicitly support recursion. But it can be extended with primitive constants (as long as they satisfy the consistency condition in Requirement 3.2 below). So, to encode inductive definitions, like `nextPrime` in our system, we use the fixpoint constant `fix`:

```
fix :: (a →  a) →  a
nextPrime = fix @(Nat→ Prime) (\f x→ if checkPrime x then x else f (x
 +1))
```

Importantly, our calculus is fully polymorphic, in the sense that type variables can be instantiated with refined types. So, the type variable of `fix` can be instantiated with the refined type `Nat → Prime` to get the desired type of `nextPrime`. Here, for emphasis, we make this instantiation explicit, but in real systems, like LIQUIDHASKELL, the refined type application is inferred.

Primes Array Example As a bigger example, consider an example where refinements are used for both error prevention and functional correctness. The function `primes n` generates an array with the first `n` prime numbers:

```
primes :: n:Nat → ArrayN Prime n
primes n = (fix go) 1 0 (new n (nextPrime 1))
  where go f i p acc = if i < n
                       then let p' = nextPrime (p+1) in
```

```
                go f (i+1) p' (set n i p' acc)
            else acc
```

Since `primes` typechecks under the safe array interface of Figure 2.1, no out-of-bounds errors will occur. At the same time, all elements of the array are `set` by a result `nextPrime` and thus `primes` returns an array of prime numbers.

2.2 The Essence of Refinement Types

The practicality of refinement types, as illustrated in the examples above, is due to the combination of three essential features:

1. ***Semantic Subtyping:*** The user does not need to provide any explicit type casts, because subtyping is implicit and semantic. For example, to type check `get 10 4 array10` (from § 2.1), the type of `4 :: {v:Int | v == 4}` is implicitly converted to `{v:Int | 0 ≤ v < 10}`

2. ***Decidability:*** The semantic casts are reduced to logical implications checked by an SMT solver. Refinement types are designed to generate decidable logical implications, to ensure predictable verification and also permit type inference [42] that makes verification practical, *e.g.* the `primes` definition requires zero annotations.

3. ***Polymorphism:*** Polymorphism on refinement types permits instantiation of type variables with any refined type. For example, the same array interface can be used to describe primes, functions with positive domains, and any other concept encoded as a refinement type.

2.3 The Design of Refinement Types

Next, we develop a minimal calculus λ_{RF} that shows how Refinement type systems enjoy the three essential features of § 2.2. λ_{RF} has four judgements that relate expressions (e), types (t),

kinds (k), predicates (p), and environments (Γ): (1) typing ($\Gamma \vdash e : t$), (2) subtyping ($\Gamma \vdash t_1 \preceq t_2$), (3) well-formedness ($\Gamma \vdash_w t : k$), and (4) implication checking ($\Gamma \vdash p_1 \Rightarrow p_2$). In § 3.3 we define the judgements in detail. Here, we present the design decisions that ensure the three essential features of refinement types.

2.3.1 Semantic Subtyping

Refinement types rely on implicit semantic subtyping, that is, type conversion (from subtypes) happens without any explicit casts and is checked semantically via logical validity. For example, in the application `get 10 4 array10` (of Fig. 2.1), the type of `4` was implicitly converted. To see how, consider an environment Γ that contains the array interface. Let $\Gamma \subseteq \{\texttt{get} : n : \texttt{Int} \to i : \texttt{Int}\{\nu : \nu < n\} \to \texttt{ArrayN}\ a\ n \to a\}$ For brevity, we ignore the requirement that i and n are natural numbers and, as in Fig. 2.1, we use $\texttt{ArrayN}\ a\ n$ as shorthand for $\texttt{Int}\{\nu : \nu < n\} \to a$. The application $(\texttt{get}\ 10)\ 4$ will type check as below, using the T-SUB rule to implicitly convert the type of the argument and the S-BASE rule to check that 4 is a valid index by checking the validity of the formula $\forall \nu.\ \nu = 4 \Rightarrow \nu < 10$.

$$\dfrac{\dfrac{\ldots}{\Gamma \vdash \texttt{get}\ 10 : \texttt{Int}\{\nu : \nu < 10\} \to \ldots} \quad \dfrac{\dfrac{}{\Gamma \vdash 4 : \texttt{Int}\{\nu : \nu = 4\}} \quad \dfrac{\forall \nu.\nu = 4 \Rightarrow \nu < 10}{\Gamma \vdash \texttt{Int}\{\nu : \nu = 4\} \preceq \texttt{Int}\{\nu : \nu < 10\}}\text{S-BASE}}{\Gamma \vdash 4 : \texttt{Int}\{\nu : \nu < 10\}}\text{T-SUB}}{\Gamma \vdash \texttt{get}\ 10\ \ 4 : \texttt{ArrayN}\ a\ 10 \to a}$$

Importantly, most refinement type systems use syntax-directed rules to destruct subtyping obligations into basic (semantic) implications. For example, in Figure 3.7 the rule S-FUN states that functions are covariant on the result and contravariant on the arguments. Thus, a refinement type system can, without any casts, decide that $a_{20} : \texttt{ArrayN}\ a\ 20$ is a suitable argument for the higher order function $\texttt{get}\ 10\ 4 : \texttt{ArrayN}\ a\ 10 \to a$ and type check the expression $\texttt{get}\ 10\ 4\ a_{20}$.

2.3.2 Decidability

As illustrated in the previous type derivation, refinement type checking essentially generates a set of verification conditions (VCs) whose validity implies type safety. Importantly, the refinement type checking rules are designed to generate VCs in the logical language used by the user-provided specifications. In general, let $\mathcal{L}$ be a logical language that contains equality and conjunction. If all the user-specified predicates belong to $\mathcal{L}$, then the VCs will be in $\mathcal{L}$ as well. In practice (*e.g.* in Liquid Haskell [50] and Flux [29]), $\mathcal{L}$ is the qualifier-free logic of equality, uninterpreted functions, and linear arithmetic (QF-EUFLIA).

To achieve this logical-language preservation, special care is taken in type checking function declarations and applications.

Function Declarations Function declarations are checked using the refinement type rule for let bindings (Rule T-LET also in Figure 3.6).

$$\frac{\Gamma \vdash e_f : t_f \qquad \Gamma \vdash_w t : k \qquad f{:}t_f, \Gamma \vdash e : t}{\Gamma \vdash \texttt{let}\ f = e_f\ \texttt{in}\ e : t}\ \text{T-LET}$$

The type checking must infer the type t_f of the function, but that could be user-annotated (*e.g.* e_f could be $e'_f : t_f$).

Importantly, the body e is checked without knowledge of the definition of f. The exact encoding of the body of the function definitions (for example, as done in Dafny [30] or Prusti [1]) requires the use of $\forall$-quantifiers in the SMT solver, thus potentially leading to undecidability. Instead, refinement types only use the refinement type specifications of functions, providing a fast but incomplete verification technique. For example, given only the specifications of `get` and `set`, and not their exact definitions, it is *not* possible to show that `get` after `set` returns the value that was set.

```
getSet :: n:Int → i:{Nat|i<n} → x:a → ArrayN a n → {v:a|x == v}
```

```
getSet n i x a = get n i (set n i x a) -- Refinement Type Error
```

Function Application For decidable type checking, refinement types use an existential type [26] to check dependent function application, *i.e.* the TApp-Exists rule below, instead of the standard type-theoretic TApp-Exact rule.

$$\frac{\Gamma \vdash f : x{:}t_x \to t \qquad \Gamma \vdash e : t_x}{\Gamma \vdash f\ e : t[e/x]}\text{TApp-Exact} \qquad \frac{\Gamma \vdash f : x{:}t_x \to t \qquad \Gamma \vdash e : t_x}{\Gamma \vdash f\ e : \exists x{:}t_x.t}\text{TApp-Exists}$$

To understand the difference, consider some expression e of type `Pos` and the identity function f

$$e : \texttt{Pos} \qquad\qquad f : x{:}\texttt{Int} \to \texttt{Int}\{v : v = x\}$$

The application $f\ e$ is typed as $\texttt{Int}\{v : v = e\}$ with the TApp-Exact rule, which has two problems. First, the information that e is positive is lost. To regain this information the system needs to re-analyze the expression e breaking compositional reasoning. Second, the arbitrary expression e enters the refinement logic potentially breaking decidability. Using the TApp-Exists rule, both of these problems are addressed. Typing first uses subtyping on f to track the actual type of the argument, thus weakening the type of f to $f : x{:}\texttt{Pos} \to \texttt{Int}\{v : v = x\}$. With this, the type of $f\ e$ becomes $\exists x{:}\texttt{Pos}.\texttt{Int}\{v : v = x\}$ preserving the information that the application argument is positive, while the variable x cannot break any carefully crafted decidability guarantees.

Knowles and Flanagan [26] introduce the existential application rule and show that it preserves the decidability and completeness of the refinement type system. An alternative approach for decidable and compositional type checking is to ensure that all the application arguments are variables by ANF transforming the original program [16]. ANF is more amicable to *implementation* as it does not require the definition of one more type form. However, ANF is more problematic for the *metatheory*, as ANF is not preserved by evaluation. Additionally, existentials

let us *synthesize* types for let-binders in a bidirectional style: when typing $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, the existential lets us eliminate x from the type synthesized for e_2, yielding a precise, algorithmic system [12]. Thus, we choose to use existential types in λ_{RF}.

2.3.3 Polymorphism

Polymorphism is a precious type abstraction [54], but combined with refinements, it can lead to imprecise or, worse, unsound systems. As an example, below we present the function `max` with four potential type signatures.

	Definition	`max`	$=$	$\lambda x\, y.\texttt{if}\ x < y\ \texttt{then}\ y\ \texttt{else}\ x$
Attempt 1:	*Monomorphism*	`max`	::	$x{:}\texttt{Int} \to y{:}\texttt{Int} \to \texttt{Int}\{\nu : x \leq \nu \wedge y \leq \nu\}$
Attempt 2:	*Unrefined Polymorphism*	`max`	::	$x{:}\alpha \to y{:}\alpha \to \alpha$
Attempt 3:	*Refined Polymorphism*	`max`	::	$x{:}\alpha \to y{:}\alpha \to \alpha\{\nu : x \leq \nu \wedge y \leq \nu\}$
λ_{RF}:	*Kinded Polymorphism*	`max`	::	$\forall \alpha{:}B.x{:}\alpha \to y{:}\alpha \to \alpha\{\nu : x \leq \nu \wedge y \leq \nu\}$

As a first attempt, we give `max` a monomorphic type, stating that the result of `max` is an integer greater than or equal to each of its arguments. This type is insufficient because it forgets any information known for `max`'s arguments. For example, if both arguments are positive, the system cannot decide that `max x y` is also positive. To preserve the argument information we give `max` a polymorphic type, as a second attempt. Now the system can deduce that `max x y` is positive, but forgets that it is also greater than or equal to both `x` and `y`. In a third attempt, we naively combine the benefits of polymorphism with refinements to give `max` a very precise type that is sufficient to propagate the arguments' properties (positivity) and `max` behavior (inequality).

Unfortunately, refinements on arbitrary type variables are dangerous for two reasons. First, the type of `max` implies that the system allows comparison of any values (including functions). Second, if refinements on type variables are allowed, then, for soundness [4], all the types that

substitute variables should be refined. For example, as detailed in §6 of [23], if a type variable is refined with `false` (*i.e.* $\alpha\{\nu : \texttt{false}\}$) and gets instantiated with an unrefined function type ($x{:}t_x \to t$), then the `false` refinement is lost and the system becomes unsound.

Base Kind when Refined To preserve the benefits of refinements on type variables, without the complications of refining function types, we introduce a kind system that separates the type variables that can be refined from the ones that cannot. To do so, we extend the standard well-formedness rule of refinement types to also perform kind checking ($\Gamma \vdash_w t : k$). Variables with the base kind B can be refined, compared, and only substituted by base, refined types. The other type variables have kind $\star$ and can only be trivially refined with `true`. With this kind system, we have a simple and convenient way to encode comparable values, and we can give `max` a polymorphic and precise type that naturally rejects non-comparable (*e.g.* function) arguments.

This simple kind system could be further stratified, *i.e.* if some base types did not support comparison, and it could be implemented via typeclass constraints, if our system contained data types. A first step towards data types is presented in Chapter Chapter 9, where we add polymorphic lists to λ_{RF}, which can be refined (for instance, to restrict the length of the list), but cannot be compared like base types. This latter restriction is needed because lists themselves can contain incomparable terms like functions.

2.4 The Soundness of Refinement Types

In this work we establish two soundness theorems for refinement types that precisely relate typing judgments $\Gamma \vdash e : t$ with the high-level goals of error prevention (type safety) and functional correctness (denotational soundness).

1. Type Safety ensures that well-typed programs do not get stuck at runtime. It says that if an expression has a type ($\emptyset \vdash e : t$) and evaluates to another expression ($e \hookrightarrow^* e'$), then either evaluation reached a value or it can take another step ($e' \hookrightarrow e''$). In λ_{RF}, we use the primitive

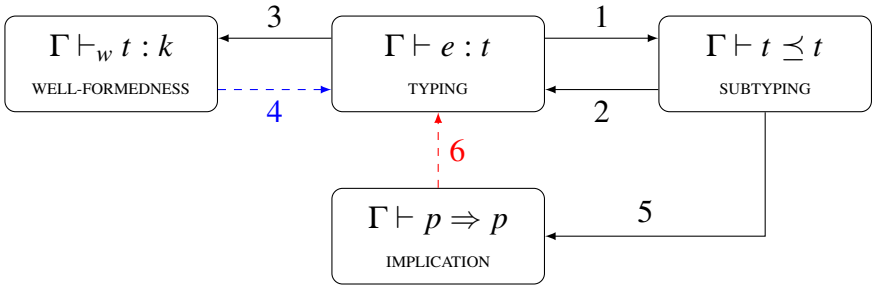

Figure 2.2: Dependencies of Typing Judgements in Refinement Types. (Dashed lines do not exist in our formalism.)

`error` to denote program errors (such as out-of-bounds indexing of Figure 2.1). The `error` primitive neither is a value nor takes a step. Thus, if an expression type checks, via type safety, we know that `error` will not be reached at runtime. Theorem 5.3 formally defines type safety, and it is proved via the preservation and progress lemmas. Type safety ensures that programs will not get stuck, but does not ensure that they satisfy their functional specifications. This is ensured by the second soundness theorem.

2. Denotational Soundness states that if an expression has a type ($\emptyset \vdash e : t$), then it belongs in the denotations of this type ($e \in [\![t]\!]$). For example, the denotation of the type `{i:Nat | i ≤ 42}` is the set of integers between 0 and 42. In § 3.3 we inductively define the denotations of each type and Theorem 5.1 formally encodes denotational soundness. Combining Lemmas Theorem 5.3 and Theorem 5.1, we see that the refined λ_{RF} type that can be checked for a specific program is preserved under evaluation, and thus the semantics of the program is preserved under evaluation.

This dissertation, for the first time, mechanizes the soundness of refinement types with semantic subtyping, existential types, and polymorphism. This mechanization turned out to be challenging for three main reasons:

Challenge 1: Circularities Figure 2.2 presents the dependencies of the four typing judgements in refinement types. As we saw in the example of § 2.3.1 (and can be confirmed in the rules defined in § 3.3), typing depends on subtyping (arrow 1) which in turn depends on

implication checking (arrow 5). Subtyping depends on typing (arrow 2; because of rule S-WIT of Figure 3.7), so typing and subtyping have a circular dependency we cannot break. Typing also depends on well-formedness (arrow 3) that checks that types, especially the ones inferred by the system, are well-formed: all the variables appearing in the refinements are bound in the type environment and refinements are of boolean type. To check the type of the refinements the system could use typing thus introducing one more dependency (arrow 4) and yet another circle. We break this dependency by using an unrefined calculus (system λ_F) that erases refinements, to check that refinements are well-typed booleans. The final potential circle is introduced when implication depends on typing (arrow 6). In § 3.3.4.3 we define implication via type denotations, but as observed by Greenberg [21], in this case, special care should be taken so that the system is monotonic and thus well-defined. To avoid this dangerous circularity we again use typing of λ_F (and not λ_{RF}) to define denotations and thus implication.

In summary, circularities in typing judgements are problematic for two reasons:

1. Circularities increase the complexity of proof mechanization. Concretely, because typing and subtyping have a circular dependency, the metatheoretical lemmas (substitution, weakening, narrowing, *etc.*) require versions for both typing and subtyping, which are proved by mutual induction. If well-formedness was also included in this circularity (arrow 4), then the complexity of the proofs would greatly increase, but would not necessarily be impossible.

2. Second, circularities are problematic because they can lead to non-well-defined systems. Concretely, Greenberg [21] describes an older refinement type system in which typing appeared in the left-hand side of subtyping and, as such, it was non-monotonic and thus not well-defined. This situation corresponds to the red arrow 6 in fig. 2.2, which would make the proof impossible due to the typing judgment occurring in a negative position in the implication judgment.

Challenge 2: Implications The second mechanization challenge was the encoding of implication. In the bibliography of refinement types, implication has been defined in three ways:

1. Using *denotations* (of types as sets of terms) defined via operational semantics [53, 15]. This encoding is more convenient when proving the soundness of the system, since implication and thus subtyping and typing, directly connect with operational semantics, making the proof of soundness more direct. However, the implementation of this encoding of implication is not realistic, since it is not decidable.

2. Using *logical implication* [42, 20]. The encoding of the implication as a logical implication is the closest to the implementation of a refinement system, where an SMT is used to check logical implications. Yet, to prove soundness, a claim should be made that logical implication checked by the SMT correctly approximates the runtime semantics of the system (*i.e.* presented in rule I-LOG of § 3.3.4.2) which has never been mechanized.

3. By *axiomatization* [27]. A final approach is to leave the implication uninterpreted and axiomatize it with all the properties required to prove soundness. This approach is the easiest to mechanize, but it is dangerous, since in the past the axioms assumed for implication were inconsistent, thus soundness was "proved with flawed premises" (as quoted from Table 1 of [44]).

Our mechanization follows a combination of the first and the third approach. We specify the interface of implication (via Requirement 3.3 of § 3.3.4.1 which is encoded as an inductive data type in the proof mechanization) to articulate the exact properties required by the soundness proof. Then, in § 3.3.4.3, we implement the implication interface using the denotational semantics of the system. This encoding has two major benefits. First, the denotational implementation ensures that our interface is consistent. Second, the development of the interface leaves room for the implementation of alternative implication "oracles", *e.g.* closer to SMT solvers. Even though we did not mechanize this alternative implementation, in § 3.3.4.2 we present how logical implications are derived from the implication judgement.

Challenge 3: Proof Complexity All the three essential features of refinement types add

complexity to the mechanization of the soundness proof. Polymorphism requires the extension of well-formedness to kind checking. Semantic subtyping makes type checking not syntax-directed (thus inversion is not trivial, cf. § 5.3) and dependent upon subtyping. In turn, the existential types required for decidability make subtyping dependent upon type checking. Due to this mutual dependency, the standard metatheoretical lemmas (substitution, weakening, narrowing, *etc.*) require versions for both typing and subtyping, which are proved by mutual induction. Thus, the combination of the three essential for refinement types features makes the metatheoretical development more complex and prone to unsoundness. Once, we have carefully broken the various circularities and eliminated potential sources of unsoundness, we get unsurprising, albeit strenuous, proofs of the soundness of refinement typing.

Acknowledgements for Chapter 2

This chapter is adapted from "Mechanizing Refinement Types" in the proceedings of the 51st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2024), by Michael Borkowski, Niki Vazou, and Ranjit Jhala. The dissertation author was the primary investigator and author of this material.

Chapter 3

The Languages λ_F and λ_{RF}

To cut the circularities in the metatheory, we formalize refinements using two calculi. The first is the *base* language λ_F: a classic System F [38] with call-by-value semantics extended with primitive `Int` and `Bool` types and operations. The second is the *refined* language λ_{RF} which extends λ_F with refinements. By using the first calculus to express the typing judgments for our refinements, we avoid making the well-formedness (in rule WF-REFN in § 3.3.1) and the implication (in type denotations of Figure 3.8) judgments mutually dependent with the typing judgments. We use the grey highlights for the extensions to λ_F required for λ_{RF}.

3.1 Syntax

We start by describing the syntax of terms and types in the two calculi.

Constants, Values and Terms Figure 3.1 summarizes the syntax of terms in both calculi. The *primitives c* include `Int` and `Bool` constants, boolean operations, the polymorphic comparison and equality, and their curried versions. *Values v* are constants, binders and λ- and type-abstractions. Finally, the *terms e* comprise values, value- and type- applications, let-binders, annotated expressions, conditionals, and runtime errors. The types in annotations are, potentially wrong, specifications written by the user and checked by the type checker.

Primitives	c	$::=$	`true` $\mid$ `false` $\mid 0,1,2,\ldots$	*booleans and integers*
		$\mid$	$\wedge, \vee, \neg, \leftrightarrow$	*boolean ops.*
		$\mid$	$\leq, =$	*polymorphic comparisons*
Values	v	$::=$	c	*primitives*
		$\mid$	x, y, ...	*variables*
		$\mid$	$\lambda x.e$	*abstractions*
		$\mid$	$\Lambda\alpha{:}k$.e	*type abstractions*
Terms	e	$::=$	v	*values*
		$\mid$	$e_1\ e_2$	*applications*
		$\mid$	$e[t]$	*type applications*
		$\mid$	`let` $x = e_1$ `in` e_2	*let-binders*
		$\mid$	$e : t$	*annotations*
		$\mid$	`if` e_0 `then` e_1 `else` e_2	*conditionals*
		$\mid$	`error`	*runtime errors*

Figure 3.1: Syntax of Primitives, Values, and Expressions.

Kinds	k	$::=$	$B \mid \star$	*base and star kind*
Predicates	p	$::=$	$\{e \mid \exists \Gamma.\Gamma \vdash_F e : \texttt{Bool}\}$	*boolean-typed terms*
Base Types	b	$::=$	`Bool` $\mid$ `Int` $\mid \alpha$	*bool, ints, and type variables*
Types	t	$::=$	$b\{\nu : p\}$	*refined base type*
		$\mid$	$x{:}t_x \to t$	*function type*
		$\mid$	$\exists x{:}t_x.t$	*existential type*
		$\mid$	$\forall \alpha{:}k.t$	*polymorphic type*
Environments	Γ	$::=$	$\varnothing \mid \Gamma, x{:}t \mid \Gamma, \alpha{:}k$	*variable and type bindings*

Figure 3.2: Syntax of Types. The gray boxes are the extensions to λ_F needed by λ_{RF}. We use τ for λ_F-only types.

Kinds & Types Figure 3.2 shows the syntax of the types, with the gray boxes indicating the extensions to λ_F required by λ_{RF}. In λ_{RF}, only base types can be refined: we do not permit refinements for functions and polymorphic types. λ_{RF} enforces this restriction using two kinds which denote types that may (B) or may not ($\star$) be refined. The (unrefined) *base* types b comprise `Int`, `Bool`, and type variables α. The simplest type is of the form $b\{\nu : p\}$ comprising a base type b and a *refinement* that restricts b to the subset of values ν that satisfy p *i.e.* for which p evaluates to `true`. We use refined base types to build up dependent function types (where the input parameter x can appear in the output type's refinement), existential and polymorphic types.

In the sequel, we write b to abbreviate $b\{\nu : \texttt{true}\}$ and call types refined with only `true` "trivially refined" types.

Refinement Erasure The reduction semantics of our polymorphic primitives are defined using an *erasure* function that returns the unrefined, λ_F version of a refined λ_{RF} type:

$$\lfloor b\{\nu : p\}\rfloor \doteq b, \quad \lfloor x{:}t_x \to t\rfloor \doteq \lfloor t_x\rfloor \to \lfloor t\rfloor, \quad \lfloor \exists x{:}t_x.t\rfloor \doteq \lfloor t\rfloor, \quad \text{and} \quad \lfloor \forall \alpha{:}k.t\rfloor \doteq \forall \alpha{:}k.\lfloor t\rfloor$$

Environments Figure 3.2 describes the syntax of typing environments Γ which contain both term variables bound to types and type variables bound to kinds. These variables may appear in types bound later in the environment. In our formalism, environments grow from right to left.

Note on Variable Representation Our metatheory requires that all variables bound in the environment are distinct. Our mechanization enforces this invariant via the locally nameless representation [2]: free and bound variables are distinct objects in the syntax, as are type and term variables. All free variables have unique names which never conflict with bound variables represented as de Bruijn indices. This eliminates the possibility of capture in substitution and the need to perform alpha-renaming during substitution. The locally nameless representation avoids technical manipulations such as index shifting by using names instead of indices for free variables (we discuss alternatives in § 1.2). To simplify the presentation of the syntax and rules, we use names for bound variables to make the dependent nature of the function arrow clear.

3.2 Dynamic Semantics

Figure 3.3 summarizes the substitution-based, call-by-value, contextual, small-step semantics for both calculi. We specify the reduction semantics of the primitives using the functions δ and δ_T.

Substitution The key difference with standard formulations is the notion of substitution for type variables at (polymorphic) type-application sites as shown in rule E-TAPP. Type

Operational Semantics $\boxed{e \hookrightarrow e'}$

$$\frac{}{c\ v \hookrightarrow \delta(c,v)}\text{E-PRIM} \quad \frac{}{c[t] \hookrightarrow \delta_T(c, \lfloor t \rfloor)}\text{E-TPRIM} \quad \frac{e \hookrightarrow e'}{e : t \hookrightarrow e' : t}\text{E-PANN} \quad \frac{}{v : t \hookrightarrow v}\text{E-ANN}$$

$$\frac{e \hookrightarrow e'}{e\ e_1 \hookrightarrow e'\ e_1}\text{E-PLAPP} \quad \frac{e \hookrightarrow e'}{v\ e \hookrightarrow v\ e'}\text{E-PRAPP} \quad \frac{}{(\lambda x.e)\ v \hookrightarrow e[v/x]}\text{E-APP}$$

$$\frac{}{(\Lambda\alpha{:}k.e)[t] \hookrightarrow e[t/\alpha]}\text{E-TAPP} \quad \frac{e \hookrightarrow e'}{e[t] \hookrightarrow e'[t]}\text{E-PTAPP}$$

$$\frac{e_x \hookrightarrow e'_x}{\texttt{let}\ x = e_x\ \texttt{in}\ e \hookrightarrow \texttt{let}\ x = e'_x\ \texttt{in}\ e}\text{E-PLET} \quad \frac{}{\texttt{let}\ x = v\ \texttt{in}\ e \hookrightarrow e[v/x]}\text{E-LET}$$

$$\frac{e \hookrightarrow e'}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2}\text{E-PIF}$$

$$\frac{}{\texttt{if true then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow e_1}\text{E-IFT} \quad \frac{}{\texttt{if false then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow e_2}\text{E-IFF}$$

Figure 3.3: The small-step semantics.

$$
\begin{array}{rl@{\qquad}rl}
\beta\{x:p\}[t_\alpha/\alpha] &\doteq \beta\{x:p[t_\alpha/\alpha]\}, \alpha \neq \beta & \mathsf{refine}(\alpha\{z:q\},p,x) &\doteq \alpha\{z:p[z/x]\wedge q\}\\
(x{:}t_x \to t)[t_\alpha/\alpha] &\doteq x{:}(t_x[t_\alpha/\alpha]) \to t[t_\alpha/\alpha] & \mathsf{refine}(\exists z{:}t_z.t,p,x) &\doteq \exists z{:}t_z.\,\mathsf{refine}(t,p,x)\\
(\exists x{:}t_x.t)[t_\alpha/\alpha] &\doteq \exists x{:}(t_x[t_\alpha/\alpha]).t[t_\alpha/\alpha] & \mathsf{refine}(x{:}t_x \to t,\_,\_) &\doteq x{:}t_x \to t\\
(\forall\beta{:}k.t)[t_\alpha/\alpha] &\doteq \forall\beta{:}k.t[t_\alpha/\alpha] & \mathsf{refine}(\forall\alpha{:}k.t,\_,\_) &\doteq \forall\alpha{:}k.t\\
\alpha\{x:p\}[t_\alpha/\alpha] &\doteq \mathsf{refine}(t_\alpha,p[t_\alpha/\alpha],x)
\end{array}
$$

Figure 3.4: Type substitution and refinement strengthening.

substitution is defined on the left of Figure 3.4, and it is standard except for the last line which defines the substitution of a type variable α in a refined type variable $\alpha\{x:p\}$ with a (potentially refined) type t_α. To do this substitution, we combine p with the type t_α by using $\mathsf{refine}(t_\alpha,p,x)$ which essentially conjoins the refinement p to the top-level refinement of a base-kinded t_α. For existential types, refine *pushes* the refinement through the existential quantifier. Function and quantified types are left unchanged as they cannot instantiate a *refined* type variable (which must be of base kind).

Primitives The function $\delta(c,v)$ evaluates the application $c\ v$ of built-in monomorphic primitives. The reductions are defined in a curried manner, *i.e.* $\leq m\ n$ evaluates to $\delta(\delta(\leq,m),n)$. Currying gives us unary relations like $m\leq$ which is a partially evaluated version of the $\leq$ relation. The function $\delta_T(c,\lfloor t\rfloor)$ specifies the reduction rules for type application on the polymorphic built-in primitives.

$$
\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
\delta(\wedge,\texttt{true}) &\doteq& \lambda x.x & \delta(\leq,m) &\doteq& m\leq & \delta_T(=,\texttt{Bool}) &\doteq& =\\
\delta(\wedge,\texttt{false}) &\doteq& \lambda x.\texttt{false} & \delta(m\leq,n) &\doteq& (m\leq n) & \delta_T(=,\texttt{Int}) &\doteq& =\\
\delta(\neg,\texttt{true}) &\doteq& \texttt{false} & \delta(=,m) &\doteq& m= & \delta_T(\leq,\texttt{Bool}) &\doteq& \leq\\
\delta(\neg,\texttt{false}) &\doteq& \texttt{true} & \delta(m=,n) &\doteq& (m=n) & \delta_T(\leq,\texttt{Int}) &\doteq& \leq
\end{array}
$$

Determinism Our soundness proof uses the determinism property of the operational semantics.

Lemma 3.1 (Determinism)**.** *For every expression e, 1) there exists at most one term e' s.t. $e \hookrightarrow e'$, 2) there exists at most one value v s.t. $e \hookrightarrow^* v$, and 3) if e is a value there is no term e' s.t. $e \hookrightarrow e'$.*

3.3 Static Semantics

The static semantics of our calculi comprise four main judgment forms: (§ 3.3.1) *well-formedness* judgments that determine when a type or environment is syntactically well-formed (in λ_F and λ_{RF}); (§ 3.3.2) *typing* judgments that stipulate that a term has a particular type in a given context (in λ_F and λ_{RF}); (§ 3.3.3) *subtyping* judgments that establish when one type can be viewed as a subtype of another (in λ_{RF}); and (§ 3.3.4) *implication* judgments that establish when one predicate implies another (in λ_{RF}). Next, we present the static semantics of λ_{RF} by describing the rules that establish each of these judgments. We use grey to highlight the antecedents and rules specific to λ_{RF}.

Co-finite Quantification We define our rules using the co-finite quantification technique of Aydemir et al. [3]. This technique enforces a small (but critical) restriction in the way fresh names are introduced in the antecedents of rules. For example, below we present the standard (on the left) and our (on the right) rules for type abstraction.

$$\frac{\alpha' \notin \Gamma \;\; \alpha' : k, \Gamma \vdash e[\alpha'/\alpha] : t[\alpha'/\alpha]}{\Gamma \vdash \Lambda\alpha{:}k.e : \forall\alpha{:}k.t}\ \text{T-ABS-EX} \qquad \frac{\forall\alpha' \notin L. \;\; \alpha' : k, \Gamma \vdash e[\alpha'/\alpha] : t[\alpha'/\alpha]}{\Gamma \vdash \Lambda\alpha{:}k.e : \forall\alpha{:}k.t}\ \text{T-TABS}$$

The standard rule T-ABS-EX requires the existence of a fresh type variable name α'. Instead, our co-finite quantification rule states that the rule holds for any name excluding a finite set of names L. As observed by Aydemir et al. [3] this rephrasing simplifies the mechanization of metatheory by eliminating the need for renaming lemmas.

3.3.1 Well-formedness

Judgments The judgment $\Gamma \vdash_w t : k$ says that the type t is well-formed in the environment Γ and has kind k. The judgment $\vdash_w \Gamma$ says that the environment Γ is well-formed, meaning that it only binds to well-formed types. Well-formedness is also used in the (unrefined) system λ_F,

where $\Gamma \vdash^{F}_{w} \tau : k$ means that the (unrefined) λ_F type τ is well-formed in environment Γ and has kind k and $\vdash_w \Gamma$ means that the free type variables of the environment Γ are bound earlier in the environment. Well-formedness is not strictly required for λ_F, but it simplifies the mechanization [43].

Rules Figure 3.5 summarizes the rules that establish the well-formedness of types and environments. Rule WF-BASE states that the two closed base types (`Int` and `Bool`, refined with `true` in λ_{RF}) are well-formed and have base kind. Similarly, rule WF-VAR says that a type variable α is well-formed with kind k so long as $\alpha{:}k$ is bound in the environment. The rule WF-REFN stipulates that a refined base type $b\{x:p\}$ is well-formed with base kind in some environment if the unrefined base type b has base kind in the same environment and if the refinement predicate p has type `Bool` in the environment augmented by binding a fresh variable to type b. Note that if $b \equiv \alpha$ then we can only form the antecedent $\Gamma \vdash_w \alpha\{x:\texttt{true}\} : B$ when $\alpha{:}B \in \Gamma$ (rule WF-VAR), which prevents us from refining star-kinded type variables. *To break a circularity* in which well-formedness judgments appear in the antecedents of typing judgments and a typing judgment appears in the antecedents of WF-REFN, we use the λ_F judgment to check that p has type `Bool`. Our rule WF-FUNC states that a function type $x{:}t_x \to t$ is well-formed with star kind in some environment Γ if both type t_x is well-formed (with any kind) in the same environment and type t is well-formed (with any kind) in the environment Γ augmented by binding a fresh variable to t_x. Rule WF-EXIS states that an existential type $\exists x{:}t_x.t$ is well-formed with some kind k in some environment Γ if both type t_x is well-formed (with any kind) in the same environment and type t is well-formed with kind k in the environment Γ augmented by binding a fresh variable to t_x. Rule WF-POLY establishes that a polymorphic type $\forall\alpha{:}k.t$ has star kind in environment Γ if the inner type t is well-formed (with any kind) in environment Γ augmented by binding a fresh type variable α to kind k. Finally, rule WF-KIND simply states that if a type t is well-formed with base kind in some environment, then it is also well-formed with star kind. This rule is required by our metatheory to convert base to star kinds in type variables.

Well-formed Type $\boxed{\Gamma \vdash_w t : k}$

$$\frac{b \in \{\texttt{Bool}, \texttt{Int}\}}{\Gamma \vdash_w b\,\{x:\texttt{true}\} : B}\text{WF-BASE} \qquad \frac{\alpha{:}k \in \Gamma}{\Gamma \vdash_w \alpha\,\{x:\texttt{true}\} : k}\text{WF-VAR} \qquad \frac{\Gamma \vdash_w t : B}{\Gamma \vdash_w t : \star}\text{WF-KIND}$$

$$\frac{\Gamma \vdash_w b\{x:\texttt{true}\} : B \quad \forall y \notin \Gamma.y{:}b, \lfloor\Gamma\rfloor \vdash_F p[y/x] : \texttt{Bool}}{\Gamma \vdash_w b\{x:p\} : B}\text{WF-REFN}$$

$$\frac{\Gamma \vdash_w t_x : k_x \quad \forall y \notin \Gamma.\ y{:}t_x, \Gamma \vdash_w t\,[y/x] : k}{\Gamma \vdash_w x{:}t_x \to t : \star}\text{WF-FUNC}$$

$$\frac{\Gamma \vdash_w t_x : k_x \quad \forall y \notin \Gamma.\ y{:}t_x, \Gamma \vdash_w t[y/x] : k}{\Gamma \vdash_w \exists x{:}t_x.t : k}\text{WF-EXIS}$$

$$\frac{\forall \alpha' \notin \Gamma.\ \alpha'{:}k, \Gamma \vdash_w t[\alpha'/\alpha] : k_t}{\Gamma \vdash_w \forall \alpha{:}k.t : \star}\text{WF-POLY}$$

Well-formed Environment $\boxed{\vdash_w \Gamma}$

$$\frac{}{\vdash_w \varnothing}\text{WFE-EMP} \qquad \frac{\Gamma \vdash_w t_x : k_x \quad \vdash_w \Gamma \quad x \notin \Gamma}{\vdash_w x{:}t_x, \Gamma}\text{WFE-BIND} \qquad \frac{\vdash_w \Gamma \quad \alpha \notin \Gamma}{\vdash_w \alpha{:}k, \Gamma}\text{WFE-TBIND}$$

Figure 3.5: Well-formedness of types and environments. The rules for λ_F exclude the gray boxes.

As for environments, rule WFE-EMP states that the empty environment is well-formed. Rule WFE-BIND says that a well-formed environment Γ remains well-formed after binding a fresh variable x to any type t_x that is well-formed in Γ. Finally, rule WFE-TBIND states that a well-formed environment remains well-formed after binding a fresh type variable to any kind.

3.3.2 Typing

The judgment $\Gamma \vdash e : t$ states that the term e has type t in the context of environment Γ. We write $\Gamma \vdash_F e : \tau$ to indicate that term e has the (unrefined) λ_F type τ in the (unrefined) context Γ. Figure 3.6 summarizes the rules that establish typing for both λ_F and λ_{RF}, with gray for the

Typing $\boxed{\Gamma \vdash e : t}$

$$\frac{}{\Gamma \vdash c : \mathsf{ty}(c)}\text{T-PRIM} \qquad \frac{x{:}t \in \Gamma \quad \Gamma \vdash_w t : k}{\Gamma \vdash x : \mathsf{self}(t, x, k)}\text{T-VAR}$$

$$\frac{\Gamma \vdash e : t \quad \Gamma \vdash_w t : k}{\Gamma \vdash e : t : t}\text{T-ANN} \qquad \frac{\Gamma \vdash_w t : k \quad \Gamma \vdash e : s \quad \Gamma \vdash s \preceq t}{\Gamma \vdash e : t}\text{T-SUB}$$

$$\frac{\Gamma \vdash e_x : t_x \quad \Gamma \vdash e : x{:}t_x \to t}{\Gamma \vdash e\ e_x : \exists x{:}t_x.\ t}\text{T-APP} \qquad \frac{\Gamma \vdash_w t_x : k_x \quad \forall y \notin \Gamma.\, y{:}t_x, \Gamma \vdash e[y/x] : t\,[y/x]}{\Gamma \vdash \lambda x.e : x{:}t_x \to t}\text{T-ABS}$$

$$\frac{\forall \alpha' \notin \Gamma. \quad \alpha'{:}k, \Gamma \vdash e[\alpha'/\alpha] : t[\alpha'/\alpha]}{\Gamma \vdash \Lambda\alpha{:}k.e : \forall\alpha{:}k.t}\text{T-TABS} \qquad \frac{\Gamma \vdash_w t : k \quad \Gamma \vdash e : \forall\alpha{:}k.s}{\Gamma \vdash e[t] : s[t/\alpha]}\text{T-TAPP}$$

$$\frac{\Gamma \vdash e_x : t_x \quad \Gamma \vdash_w t : k \quad \forall y \notin \Gamma.\, y{:}t_x, \Gamma \vdash e[y/x] : t\,[y/x]}{\Gamma \vdash \mathtt{let}\ x = e_x\ \mathtt{in}\ e : t}\text{T-LET}$$

$$\frac{\Gamma \vdash e : \mathtt{Bool}\,\{x{:}p\} \quad \Gamma \vdash_w t : k \quad \forall y \notin \Gamma.\, y{:}\mathtt{Bool}\{x{:}p \wedge x\}, \Gamma \vdash e_1 : t \quad \forall y \notin \Gamma.\, y{:}\mathtt{Bool}\{x{:}p \wedge \neg x\}, \Gamma \vdash e_2 : t}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : t}\text{T-IF}$$

Figure 3.6: Typing rules. The judgment $\Gamma \vdash_F e : \tau$ is defined by excluding the gray boxes.

λ_{RF} extensions.

Typing Primitives The type of a built-in primitive c is given by the function $\mathsf{ty}(c)$, which is defined for every constant of our system. Below we present essential examples of the $\mathsf{ty}(c)$ definition.

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{ty}(\mathtt{true}) & \doteq & \mathtt{Bool}\{x{:}x = \mathtt{true}\} & \mathsf{ty}(\wedge) & \doteq & x{:}\mathtt{Bool} \to y{:}\mathtt{Bool} \to \mathtt{Bool}\{v{:}v = x \wedge y\} \\
\mathsf{ty}(3) & \doteq & \mathtt{Int}\{x{:}x = 3\} & \mathsf{ty}(\leq) & \doteq & \forall\alpha{:}B.x{:}\alpha \to y{:}\alpha \to \mathtt{Bool}\{v{:}v = (x \leq y)\} \\
\mathsf{ty}(m\leq) & \doteq & y{:}\mathtt{Int} \to \mathtt{Bool}\{v{:}v = (m \leq y)\} & \mathsf{ty}(=) & \doteq & \forall\alpha{:}B.x{:}\alpha \to y{:}\alpha \to \mathtt{Bool}\{v{:}v = (x = y)\}
\end{array}$$

We note that the $=$ used in the refinements is the polymorphic equals with type applications elided. Further, we use $m\leq$ to represent an arbitrary member of the infinite family of primitives $0\leq, 1\leq, 2\leq, \ldots$. For λ_F we erase the refinements using $\lfloor \mathsf{ty}(c) \rfloor$. The rest of the definition is similar.

Our choice to make the typing and reduction of constants external to our language, *i.e.* given by the functions $\mathsf{ty}(c)$ and $\delta(c)$, makes our system easily extensible with further constants, including a terminating `fix` constant to encode induction. The requirement, for soundness, is that these two functions together satisfy the following four conditions.

Requirement 3.2. *(Primitives) For every primitive c,*

1. *If* $\mathsf{ty}(c) = b\{x : p\}$*, then* $\varnothing \vdash_w \mathsf{ty}(c) : B$ *and* $\varnothing \vdash \texttt{true} \Rightarrow p[c/x]$.
2. *If* $\mathsf{ty}(c) = x{:}t_x \to t$ *or* $\mathsf{ty}(c) = \forall\alpha{:}k.t$*, then* $\varnothing \vdash_w \mathsf{ty}(c) : \star$.
3. *If* $\mathsf{ty}(c) = x{:}t_x \to t$*, then for all* v_x *such that* $\varnothing \vdash v_x : t_x$*,* $\varnothing \vdash \delta(c, v_x) : t[v_x/x]$.
4. *If* $\mathsf{ty}(c) = \forall\alpha{:}k.t$*, then for all* t_α *such that* $\varnothing \vdash_w t_\alpha : k$*,* $\varnothing \vdash \delta_T(c, t_\alpha) : t[t_\alpha/\alpha]$.

Theorem 3 of [51] proves that a terminating `fix` constant satisfies requirement 3.2.

To type constants, rule T-PRIM gives the type $ty(c)$ to any built-in primitive c, in any context. The typing rules for boolean and integer constants are included in T-PRIM.

Typing Variables with Selfification Rule T-VAR establishes that any variable x that appears as $x{:}t$ in environment Γ can be given the *selfified* type [35] $\mathsf{self}(t, x, k)$ provided that $\Gamma \vdash_w t : k$. This rule is crucial in practice, to enable path-sensitive "occurrence" typing [49], where the types of variables are refined by control-flow guards. For example, suppose we want to establish $\alpha{:}B \vdash (\lambda x.x) : x{:}\alpha \to \alpha\{y : x = y\}$, and not just $\alpha{:}B \vdash (\lambda x.x) : \alpha \to \alpha$. The latter would result if T-VAR merely stated that $\Gamma \vdash x : t$ whenever $x{:}t \in \Gamma$. Instead, we strengthen the T-VAR rule to be *selfified*. Informally, to get information about x into the refinement level, we need to say that x is constrained to elements of type α that are equal to x itself. In order to express the

exact type of variables, below we define the "selfification" function that strengthens a refinement with the condition that a value is equal to itself. Since abstractions do not admit equality, we only selfify the base types and the existential quantifications of them.

$$\mathsf{self}(\exists z{:}t_z.t,x,k) \doteq \exists z{:}t_z.\mathsf{self}(t,x,k) \qquad \mathsf{self}(b\{z:p\},x,B) \doteq b\{z:p \wedge z = x\}$$
$$\mathsf{self}(x{:}t_x \to t,\_,\_) \doteq x{:}t_x \to t \qquad \mathsf{self}(b\{z:p\},x,\star) \doteq b\{z:p\}$$
$$\mathsf{self}(\forall\,\alpha{:}k.t,\_,\_) \doteq \forall\,\alpha{:}k.t$$

Typing Applications with Existentials Our rule T-APP states the conditions for typing a term application $e\ e_x$. Under the same environment, we must be able to type e at some function type $x{:}t_x \to t$ and e_x at t_x. Then we can give $e\ e_x$ the existential type $\exists x{:}t_x.t$. The use of existential types in rule T-APP is one of the distinctive features of our language and was introduced by Knowles and Flanagan [26]. As overviewed in § 2.3.2, we chose this form of T-APP over the conventional form of $\Gamma \vdash e\ e_x : t[e_x/x]$ because our version prevents the substitution of arbitrary expressions (*e.g.* functions and type abstractions) into refinements. As an alternative, we could have used ANF (A-Normal Form [16]), but our metatheory would be more complex since ANF is not preserved under the small step operational semantics.

Other Typing Rules Rule T-ABS says that we can type a lambda abstraction $\lambda x.e$ at a function type $x{:}t_x \to t$ whenever t_x is well-formed and the body e can be typed at t in the environment augmented by binding a fresh variable to t_x. Our rule T-TAPP states that whenever a term e has polymorphic type $\forall\,\alpha{:}k.s$, then for any well-formed type t with kind k, we can give the type $s[t/\alpha]$ to the type application $e[t]$. For the λ_F variant of T-TAPP, we erase the refinements (via $\lfloor t \rfloor$) before checking well-formedness and performing the substitution. The rule T-TABS establishes that a type-abstraction $\Lambda\alpha{:}k.e$ can be given a polymorphic type $\forall\,\alpha{:}k.t$ in some Γ whenever e can be given the well-formed type t in Γ augmented by binding a fresh type variable to kind k. Next, rule T-LET states that an expression $\texttt{let}\ x = e_x\ \texttt{in}\ e$ has type t in some environment whenever t is well-formed, e_x can be given some type t_x, and the body e can be given type t in

the environment augmented by binding a fresh variable to t_x. Rule T-ANN establishes that an explicit annotation $e : t$ indeed has type t when the underlying e has type t and t is well-formed. The λ_F version of the rule erases the refinements and uses $\lfloor t \rfloor$. Rule T-IF states that a conditional expression `if` e `then` e_1 `else` e_2 has the type t when the guard e can be given type `Bool` refined by p and e_1 (*resp.* e_2) can be given type t in the environment Γ augmented by the knowledge we have about the type and semantics of the guard e. The extension of the environment Γ with a fresh variable that captures the semantics of the guard when checking the two paths is critical to permit path-sensitive reasoning. Finally, rule T-SUB tells us that we can exchange a subtype s for a supertype t in a judgment $\Gamma \vdash e : t$ provided t is well-formed and $\Gamma \vdash s \preceq t$, which we present next.

3.3.3 Subtyping

The *subtyping* judgment $\Gamma \vdash s \preceq t$, defined in Figure 3.7, stipulates that the type s is a subtype of the type t in the environment Γ and is used in the subsumption typing rule T-SUB (of Figure 3.6).

Subtyping Rules The rule S-FUN states that one function type $x_1{:}t_{x1} \to t_1$ is a subtype of another function type $x_2{:}t_{x2} \to t_2$ in a given environment Γ when both t_{x2} is a subtype of t_{x1} and t_1 is a subtype of t_2 when we augment Γ by binding a fresh variable to type t_{x2}. As usual, function subtyping is contravariant in the input type and covariant in the outputs. Rules S-BIND and S-WIT establish subtyping for existential types [26], *resp.* when the existential appears on the left or right. Rule S-BIND allows us to exchange a universal quantifier (a variable bound to some type t_x in the environment) for an existential quantifier. If we have a judgment of the form $y{:}t_x, \Gamma \vdash t[y/x] \preceq t'$ where y does *not* appear free in either t' or in the context Γ, then we can conclude that $\exists x{:}t_x.t$ is a subtype of t'. Rule S-WIT states that if type t is a subtype of $t'[v_x/x]$ for some value v_x of type t_x, then we can discard the specific *witness* for x and quantify existentially to obtain that t is a subtype of $\exists x{:}t_x.t'$. Rule S-POLY states that one polymorphic type $\forall \alpha{:}k.t_1$ is

Subtyping $\boxed{\Gamma \vdash s \preceq t}$

$$\frac{\forall y \notin \Gamma. \quad y{:}b, \Gamma \vdash p_1[y/x] \Rightarrow p_2[y/x]}{\Gamma \vdash b\{x{:}p_1\} \preceq b\{x{:}p_2\}}\ \text{S-BASE}$$

$$\frac{\Gamma \vdash t_{x2} \preceq t_{x1} \quad \forall y \notin \Gamma. \quad y{:}t_{x2}, \Gamma \vdash t_1[y/x] \preceq t_2[y/x]}{\Gamma \vdash x{:}t_{x1} \to t_1 \preceq x{:}t_{x2} \to t_2}\ \text{S-FUN}$$

$$\frac{\Gamma \vdash v_x : t_x \quad \Gamma \vdash t \preceq t'[v_x/x]}{\Gamma \vdash t \preceq \exists x{:}t_x.t'}\ \text{S-WIT} \qquad \frac{\forall y \notin \mathsf{free}(t) \cup \Gamma. \quad y{:}t_x, \Gamma \vdash t[y/x] \preceq t'}{\Gamma \vdash \exists x{:}t_x.t \preceq t'}\ \text{S-BIND}$$

$$\frac{\forall \alpha' \notin \Gamma. \quad \alpha'{:}k, \Gamma \vdash t_1[\alpha'/\alpha] \preceq t_2[\alpha'/\alpha]}{\Gamma \vdash \forall \alpha{:}k.t_1 \preceq \forall \alpha{:}k.t_2}\ \text{S-POLY}$$

Figure 3.7: Subtyping Rules.

a subtype of another polymorphic type $\forall \alpha{:}k.t_2$ in some environment Γ, when t_1 is a subtype of t_2 in the environment Γ augmented by binding a fresh type variable to kind k.

Refinements enter the scene in the rule S-BASE which specifies that a refined base type $b\{x{:}p_1\}$ is a subtype of another $b\{x{:}p_2\}$ in context Γ when p_1 *implies* p_2 in the environment Γ augmented by binding a fresh variable to the unrefined type b.

3.3.4 Implication

The *implication* judgment $\Gamma \vdash p_1 \Rightarrow p_2$ states that the implication $p_1 \Rightarrow p_2$ holds under the assumptions captured by the context Γ. In refinement type implementations [48, 50], this relation is implemented as an external automated (usually SMT) solver. Since external solvers are not easy to encode in mechanized proofs, we follow an approach that decouples the mechanization from the implementation. Concretely, first we define the interface of the implication (§ 3.3.4.1) that precisely captures all the requirements that the implication judgment should satisfy to establish the soundness of λ_{RF}. Then, we define two alternative implementations of the interface: a logical

implementation (§ 3.3.4.2) that is used in refinement type implementations and a denotational implementation (§ 3.3.4.3) that we used to complete our mechanized proof.

3.3.4.1 Implication's Interface

In our mechanization, following Lehmann and Tanter [27], we encode implication as an axiomatized judgment that satisfies the requirements below.

Requirement 3.3 (Implication Interface)**.** *The implication relation satisfies the below statements:*

1. *(Reflexivity)* $\Gamma \vdash p \Rightarrow p$.
2. *(Transitivity) If* $\Gamma \vdash p_1 \Rightarrow p_2$ *and* $\Gamma \vdash p_2 \Rightarrow p_3$, *then* $\Gamma \vdash p_1 \Rightarrow p_3$.
3. *(Faithfulness)* $\Gamma \vdash p \Rightarrow \texttt{true}$.
4. *(Introduction) If* $\Gamma \vdash p_1 \Rightarrow p_2$ *and* $\Gamma \vdash p_1 \Rightarrow p_3$, *then* $\Gamma \vdash p_1 \Rightarrow p_2 \wedge p_3$.
5. *(Conjunction)* $\Gamma \vdash p_1 \wedge p_2 \Rightarrow p_1$ *and* $\Gamma \vdash p_1 \wedge p_2 \Rightarrow p_2$.
6. *(Repetition)* $\Gamma \vdash p_1 \wedge p_2 \Rightarrow p_1 \wedge p_1 \wedge p_2$.
7. *(Evaluation) If* $p_1 \hookrightarrow^* p_2$, *then* $\Gamma \vdash p_1 \Rightarrow p_2$ *and* $\Gamma \vdash p_2 \Rightarrow p_1$.
8. *(Narrowing) If* $\Gamma_1, x{:}t_x, \Gamma_2 \vdash p_1 \Rightarrow p_2$ *and* $\Gamma_2 \vdash s_x \preceq t_x$, *then* $\Gamma_1, x{:}s_x, \Gamma_2 \vdash p_1 \Rightarrow p_2$.
9. *(Weaken) If* $\Gamma_1, \Gamma_2 \vdash p_1 \Rightarrow p_2$, $a, x \notin \Gamma$, *then* $\Gamma_1, x{:}t_x, \Gamma_2 \vdash p_1 \Rightarrow p_2$ *and* $\Gamma_1, a{:}k, \Gamma_2 \vdash p_1 \Rightarrow p_2$.
10. *(Subst I) If* $\Gamma_1, x{:}t_x, \Gamma_2 \vdash p_1 \Rightarrow p_2$ *and* $\Gamma_2 \vdash v_x : t_x$, *then* $\Gamma_1[v_x/x], \Gamma_2 \vdash p_1[v_x/x] \Rightarrow p_2[v_x/x]$.
11. *(Subst II) If* $\Gamma_1, a{:}k, \Gamma_2 \vdash p_1 \Rightarrow p_2$ *and* $\Gamma_2 \vdash_w t : k$, *then* $\Gamma_1[t/a], \Gamma_2 \vdash p_1[t/a] \Rightarrow p_2[t/a]$.
12. *(Strengthening) If* $y{:}b\{x{:}q\}, \Gamma \vdash p_1 \Rightarrow p_2$, *then* $y{:}b, \Gamma \vdash q[y/x] \wedge p_1 \Rightarrow q[y/x] \wedge p_2$.

This interface precisely explicates the requirements of the implication checker to establish the soundness of the entire refinement type system. The first six statements are standard properties of implication. Evaluation is used to prove that built-in constants satisfy the Requirement 3.2 and the rest, as captured by their name, are required to prove the narrowing (5.10), weakening (5.9), substitution (5.8) lemmas hold in λ_{RF}.

Our requirements are very similar to Assumption 1 of [26]. Our Strengthening and Subst II cases are required for polymorphism, thus they do not appear in Knowles and Flanagan [26]'s assumption. Instead, they require Consistency and Exact Quantification. We do not require Exact Quantification since our relation captures the minimum requirements to prove soundness. Instead of explicitly requiring Consistency, in § 3.3.4.3 we define (and mechanize) an implementation, *i.e.* inhabitant, of the interface thus show our assumptions are consistent.

3.3.4.2 Logical Implementation (non-mechanized)

The logical implementation of $\Gamma \vdash p_1 \Rightarrow p_2$ checks that the logical implication $p_1 \Rightarrow p_2$ is valid assuming the refinements of the base types in Γ:

$$\frac{\models_{\texttt{LOGIC}} \wedge\{p[x/\nu] \mid x{:}b\{\nu : p\} \in \Gamma\} \Rightarrow p_1 \Rightarrow p_2}{\Gamma \vdash p_1 \Rightarrow p_2}\text{I-LOG}$$

This encoding is imprecise, since some information is ignored from the environment Γ, but when the language of refinements is decidable, implication checking is also decidable and can be efficiently checked by an SMT solver. LIQUIDHASKELL, for example, uses this encoding to reduce type checking to decidable implications checked by Z3 [13], while the soundness of this implementation (concretely statement 7 of Requirement 3.3) is hinted by Theorem 2 of [51]. Chen [10] defines a mechanization of a refinement type system in Agda that uses a similar encoding of implication where logical implications are checked using Agda's logic.

$$
\begin{array}{l}
[\![b\{x:p\}]\!] \doteq \{v \mid \varnothing \vdash_F v : b \wedge p[v/x] \hookrightarrow^* \texttt{true}\} \\
[\![x{:}t_x \to t]\!] \doteq \{v \mid \varnothing \vdash_F v : \lfloor t_x \rfloor \to \lfloor t \rfloor \wedge (\forall v_x \in [\![t_x]\!]. v\ v_x \hookrightarrow^* v' \text{ s.t. } v' \in [\![t[v_x/x]]\!]\} \\
[\![\exists x{:}t_x.t]\!] \doteq \{v \mid (\varnothing \vdash_F v : \lfloor t \rfloor) \wedge (\exists v_x \in [\![t_x]\!]. v \in [\![t[v_x/x]]\!]\} \\
[\![\forall \alpha{:}k.t]\!] \doteq \{v \mid (\varnothing \vdash_F v : \forall \alpha{:}k. \lfloor t \rfloor) \wedge (\forall t_\alpha. (\varnothing \vdash_w t_\alpha : k) \Rightarrow v[t_\alpha] \hookrightarrow^* v' \text{ s.t. } v' \in [\![t[t_\alpha/\alpha]]\!]\}
\end{array}
$$

$$
[\![\Gamma]\!] \doteq \{\theta \mid \forall (x:t) \in \Gamma. \theta(x) \in [\![\theta \cdot t]\!] \wedge \forall (\alpha : k) \in \Gamma. \varnothing \vdash_w \theta(\alpha) : k\}.
$$

Figure 3.8: Denotations of Types and Environments.

3.3.4.3 Denotational Implementation (mechanized)

The denotational implementation of $\Gamma \vdash p_1 \Rightarrow p_2$ checks that if p_1 evaluates to `true`, so does p_2.

$$
\frac{\forall \theta \in [\![\Gamma]\!].\ \theta \cdot p_1 \hookrightarrow^* \texttt{true} \Rightarrow \theta \cdot p_2 \hookrightarrow^* \texttt{true}}{\Gamma \vdash p_1 \Rightarrow p_2}\text{I-DEN}
$$

The refinements p_1 and p_2 are boolean expressions, so evaluation uses the operational semantics of Figure 3.3. But, they are open expressions with variables bound in Γ, so before evaluation we apply the closing substitution θ that belongs to the denotation of Γ, as defined next.

Closing Substitutions A *closing substitution* is a sequence of value bindings to variables: $\theta = (x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, \alpha_1 \mapsto t_1, \ldots, \alpha_m \mapsto t_m)$ with all x_i, α_j distinct. We write $\theta(x)$ to refer to v_i if $x = x_i$ and we use $\theta(\alpha)$ to refer to t_j if $\alpha = \alpha_j$. We define $\theta \cdot t$ to be the type derived from t by substituting for all variables in θ: $\theta \cdot t \doteq t[v_1/x_1] \cdots [v_n/x_n][t_1/\alpha_1] \cdots [t_m/\alpha_m]$.

Denotational Semantics Figure 3.8 defines the denotations of types and environments. Following Flanagan [15], each closed type has a denotation $[\![t]\!]$ containing the set of closed values of the appropriate base type which satisfy the type's refinement predicate. (The denotation of a type variable α is not defined as we only require denotations for closed types.) We lift the notion of denotations to environments $[\![\Gamma]\!]$ as the set of closing substitutions, *i.e.* value and type bindings for the variables in Γ, such that the values respect the denotations of the respective Γ-bound types

and the types are well-formed with respect to the corresponding kinds.

Revisiting rule I-DEN The premise of the rule I-DEN quantifies over all closing substitutions in the denotations of the typing environment (*i.e.* $\forall \theta \in [\![\Gamma]\!]$). This quantification has two consequences.

First, the environment denotation appears in a negative position on the premise of the rule. Inspecting Figure 3.8, the environment denotation uses the type denotation, which in turn uses type checking, thus rendering a *potential circularity between type and implication checking* (arrow 6 of Figure 2.2). Because of the negative occurrence, this mutual dependency would lead to a non-monotonic and thus non-well defined system. To break this circularity, we use λ_F's type checking in the definition of type denotations.

Second, the quantification is over all closing substitutions which are infinite. For example, a typing environment that binds x to an integer (*i.e.* $x{:}\texttt{Int} \in \Gamma$) has infinitely many closing substitutions mapping x to a different integer. Thus, the denotational implementation cannot be used to implement a decidable type checker. On the positive side, the denotational implementation connects implication checking to the operational semantics thus it is amicable to mechanization. Concretely, we proved (§ 7.2) that the denotational implementations satisfies the statements of Requirement 3.3.

Acknowledgements for Chapter 3

This chapter is adapted from "Mechanizing Refinement Types" in the proceedings of the 51st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2024), by Michael Borkowski, Niki Vazou, and Ranjit Jhala. The dissertation author was the primary investigator and author of this material.

Chapter 4

λ_F Soundness

Next, we present the metatheory of the underlying (unrefined) λ_F. Even though it follows the textbook techniques of Pierce [38], it is a convenient stepping stone *towards* the metatheory for (refined) λ_{RF}. In addition, the soundness results for λ_F are used *for* our full metatheory, as our well-formedness judgments require the refinement predicate to have the λ_F type `Bool` thereby avoiding the circularity of using a regular typing judgment in the antecedents of the well-formedness rules. The light grey boxes in Figure 5.1 show the high level outline of the metatheory for λ_F which provides a miniaturized model for λ_{RF} but without the challenges of subtyping and existentials. Next, we describe the top-level type safety result, how it is decomposed into progress (Lemma 4.2) and preservation (Lemma 4.6) lemmas, and the various technical results that support the lemmas.

Because the syntax of λ_F is identical to the syntax of λ_{RF}, we have the slight complication that refinements still appear in our terms. Therefore, in this chapter we continue to use $t, t', \ldots$ in our definitions and theorems to denote the refined types of λ_{RF} that will be used in type annotations $e : t$ and type applications $e[t]$. To avoid confusion in the development of the unrefined metatheory, we will use $\tau, \tau', \ldots$ for λ_F types.

Well-formed (Unrefined) Types $\boxed{\Gamma \vdash^F_w \tau : k}$

$$\frac{b \in \{\texttt{Bool}, \texttt{Int}\}}{\Gamma \vdash^F_w b : B}\text{WFFT-BASIC} \quad \frac{\alpha : k \in \Gamma}{\Gamma \vdash^F_w \alpha : k}\text{WFFT-VAR} \quad \frac{\Gamma \vdash^F_w \tau : B}{\Gamma \vdash^F_w \tau : \star}\text{WFFT-KIND}$$

$$\frac{\Gamma \vdash^F_w \tau_x : k_x \quad \Gamma \vdash^F_w \tau : k}{\Gamma \vdash^F_w \tau_x \to \tau : \star}\text{WFFT-FUNC} \quad \frac{\forall \alpha' \notin \Gamma.\ \ \alpha' : k, \Gamma \vdash^F_w \tau[\alpha'/\alpha] : k_\tau}{\Gamma \vdash^F_w \forall \alpha{:}k.\, \tau : \star}\text{WFFT-POLY}$$

Figure 4.1: Well-formedness of λ_F types.

4.1 Static Semantics

The small-step semantics for λ_F are identical to those of λ_{RF} (because the syntax is unchanged), but the well-formedness and typing rules consist of those in Figures 3.5 and 3.6 with the parts in gray erased. For clarity, and to make this chapter self-contained, we present the λ_F rules for well-formedness in Figure 4.1 and the rules for typing in Figure 4.2.

4.2 Metatheory for λ_F

The main type safety theorem for λ_F states that a well-typed term does not get stuck: *i.e.* either evaluates to a value or can step to another term (progress) of the same type (preservation). The judgment $\Gamma \vdash_F e : \tau$ is defined in fig. 4.2, and for clarity we use τ for λ_F types and t for the λ_{RF} types that appear in user annotations and in type applications.

Theorem 4.1. *(Type Safety of λ_F)*

1. *(Type Safety) If $\varnothing \vdash e : \tau$ and $e \hookrightarrow^* e'$, then e' is a value or $e' \hookrightarrow e''$ for some e''.*

2. *(No Error) If $\varnothing \vdash e : \tau$ and $e \hookrightarrow^* e'$, then $e' \neq \texttt{error}$.*

Proof. (1) We proceed by induction on the number of steps in $e \hookrightarrow^* e'$. There are two cases for $e \hookrightarrow^* e'$: either $e = e'$ or there exists a term e_1 such that $e \hookrightarrow e_1 \hookrightarrow^* e'$. In the former case we

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

$$\frac{}{\Gamma \vdash_F c : \lfloor \mathsf{ty}(c) \rfloor}\text{FT-PRIM} \quad \frac{x{:}\tau \in \Gamma \quad \Gamma \vdash_w^F \tau : k}{\Gamma \vdash_F x : \tau}\text{FT-VAR} \quad \frac{\Gamma \vdash_F e : \lfloor t \rfloor \quad \Gamma \vdash_w^F \lfloor t \rfloor : k}{\Gamma \vdash_F e : t : \lfloor t \rfloor}\text{FT-ANN}$$

$$\frac{\Gamma \vdash_F e_x : \tau_x \quad \Gamma \vdash_F e : \tau_x \to \tau}{\Gamma \vdash_F e\ e_x : \tau}\text{FT-APP} \quad \frac{\Gamma \vdash_w^F \tau_x : k_x \quad \forall y \notin \Gamma.y{:}\tau_x, \Gamma \vdash e[y/x] : \tau}{\Gamma \vdash \lambda x.e : \tau_x \to \tau}\text{FT-ABS} \quad \frac{\Gamma \vdash_w^F \lfloor t \rfloor : k \quad \Gamma \vdash e : \forall \alpha{:}k.\tau'}{\Gamma \vdash e[t] : \tau'[\lfloor t \rfloor/\alpha]}\text{FT-TAPP}$$

$$\frac{\forall \alpha' \notin \Gamma. \quad \alpha'{:}k, \Gamma \vdash e[\alpha'/\alpha] : \tau[\alpha'/\alpha]}{\Gamma \vdash \Lambda\alpha{:}k.e : \forall \alpha{:}k.\tau}\text{FT-TABS}$$

$$\frac{\Gamma \vdash e_x : \tau_x \quad \forall y \notin \Gamma.y{:}\tau_x, \Gamma \vdash e[y/x] : \tau}{\Gamma \vdash \mathtt{let}\ x = e_x\ \mathtt{in}\ e : \tau}\text{FT-LET} \quad \frac{\Gamma \vdash e : \mathtt{Bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau}\text{FT-IF}$$

Figure 4.2: Unrefined typing rules.

conclude immediately by the Progress Lemma (4.2 below). In the latter case, $\varnothing \vdash_F e_1 : \tau$ by the Preservation Lemma (4.6). Then by the inductive hypothesis applied to the reduction sequence $e_1 \hookrightarrow^* e'$, we conclude that either e' is a value or $e' \hookrightarrow e''$ for some e'' as desired.

(2) The second statement follows immediately from the first: we know that either e' is a value and cannot be `error` or e' can take a step. But no rule in Figure 3.3 can be applied to reduce `error`. □

As mentioned in the proof above, we prove type safety by induction on the length of the sequence of steps comprising $e \hookrightarrow^* e'$, using the preservation and progress lemmas.

4.2.1 Progress

The progress lemma says a well-typed term is a value or steps to some other term.

Lemma 4.2. *(Progress) If $\varnothing \vdash_F e : \tau$, then e is a value or $e \hookrightarrow e'$ for some e'.*

Proof. We proceed by induction of the structure of $\varnothing \vdash_F e : \tau$. In the cases of rule FT-PRIM, FT-VAR, FT-ABS, or FT-TABS, e is a value.

Case FT-APP: We have $\varnothing \vdash_F e : \tau$ where $e \equiv e_1\ e_2$. Inverting, we have that there exists some type τ_2 such that $\varnothing \vdash_F e_1 : \tau_2 \to \tau$ and $\varnothing \vdash_F e_2 : \tau_2$. We split on five possible cases for the structure of e_1 and e_2. First, suppose $e_1 \equiv \lambda x.e_0$ and e_2 is a value. Then by rule E-APP, $e \equiv \lambda x.e_0\ e_2 \hookrightarrow e_0[e_2/x]$. Second, suppose $e_1 \equiv \lambda x.e_0$ and e_2 is not a value. Then by the inductive hypothesis, there exists a term e_2' such that $e_2 \hookrightarrow e_2'$. Then by rule E-PRAPP $e \equiv \lambda x.e_0\ e_2 \hookrightarrow \lambda x.e_0\ e_2'$. Third, suppose $e_1 \equiv c$, a built-in primitive and e_2 is a value. Then by rule E-PRIM, $e \equiv c\ e_2 \hookrightarrow \delta(c, e_2)$, which is well-defined by the Primitives Lemma (4.5). Fourth, suppose $e_1 \equiv c$ and e_2 is not a value. Then by the inductive hypothesis, there exists a term e_2' such that $e_2 \hookrightarrow e_2'$. Then by rule E-PRAPP $e \equiv c\ e_2 \hookrightarrow c\ e_2'$. Finally, by the Canonical Forms Lemma (4.3), e_1 cannot be any other value, so it must not be a value. Then

by the inductive hypothesis, there is a term e_1' such that $e_1 \hookrightarrow e_1'$. Then by rule E-PLAPP, $e \equiv e_1\ e_2 \hookrightarrow e_1'\ e_2$.

Case FT-TAPP: We have $\varnothing \vdash_F e : \tau$ where $e \equiv e_1[t]$ and $\tau \equiv \sigma[\lfloor t \rfloor/\alpha]$. Inverting, we have that $\varnothing \vdash_F e_1 : \forall\,\alpha{:}k.\,\sigma$. We split on three cases for the structure of e_1. First, suppose $e_1 \equiv \Lambda\alpha'{:}k'.e_0$. Then by rule E-TAPP, $e \equiv \Lambda\alpha'{:}k'.e_0[t] \hookrightarrow e_0[t/\alpha']$. Second, suppose $e_1 \equiv c$, a built-in primitive. Then by rule E-TPRIM, $e \equiv c[t] \hookrightarrow \delta_T(c, \lfloor t \rfloor)$, which is well-defined by the Primitives Lemma (4.5). Finally, by the Canonical Forms Lemma (4.3), e_1 cannot be any other form of value, so it must not be a value. Then by the inductive hypothesis, there is a term e_1' such that $e_1 \hookrightarrow e_1'$. Then by rule E-PTAPP $e \equiv e_1[t] \hookrightarrow e_1'[t]$.

Case FT-LET: We have $\varnothing \vdash_F e : \tau$ where $e \equiv \texttt{let}\ x = e_1\ \texttt{in}\ e_2$. Inverting, we have that $\varnothing \vdash_F e_1 : \tau_1$ for some type τ_1. By the inductive hypothesis, either e_1 is a value or there is a term e_1' such that $e_1 \hookrightarrow e_1'$. In the former case, rule E-LET gives us $e \equiv \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \hookrightarrow e_2[e_1/x]$. In the latter case, by rule E-PLET, $e \equiv \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \hookrightarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2$.

Case FT-ANN: We have $\varnothing \vdash_F e : \tau$ where $e \equiv e_1 : t$. Inverting, we have the $\varnothing \vdash_F e_1 : \tau$ and $\tau = \lfloor t \rfloor$. By the inductive hypothesis, either e_1 is a value or there is a term e_1' such that $e_1 \hookrightarrow e_1'$. In the former case, by rule E-ANN, $e \equiv e_1 : t \hookrightarrow e_1$. In the latter case, rule E-PANN gives us $e \equiv e_1 : t \hookrightarrow e_1' : t$.

Case FT-IF: We have $\varnothing \vdash_F e : \tau$ where $e \equiv \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. Inverting, we have that $\varnothing \vdash_F e : \texttt{Bool}$. By the inductive hypothesis, either e is a value or there is a term e' such that $e \hookrightarrow e'$. In the former case, by the Canonical Forms Lemma (4.3), we have that $e = \texttt{true}$ or $e = \texttt{false}$, and so either $e \equiv \texttt{if true then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow e_1$ by rule E-IFT or (respectively) $e \equiv \texttt{if false then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow e_2$ by rule E-IFF. In the latter case, by rule E-PIF, $e \equiv \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2$.

□

The proof of progress given above requires a *Canonical Forms* lemma (lemma 4.3) which describes the shape of well-typed values and some key properties about the built-in *Primitives* (lemma 4.5). We also implicitly use an *Inversion of Typing* lemma (lemma 4.4) which describes the shape of the type of well-typed terms and its subterms. For λ_F, unlike λ_{RF}, this lemma is trivial because the typing relation is syntax-directed.

Lemma 4.3. *(Canonical Forms)*

1. *If* $\varnothing \vdash_F v : \texttt{Bool}$*, then* $v = \texttt{true}$ *or* $v = \texttt{false}$.
2. *If* $\varnothing \vdash_F v : \texttt{Int}$*, then v is an integer constant.*
3. *If* $\varnothing \vdash_F v : \tau \to \tau'$*, then either* $v = \lambda x.e$ *or* $v = c$*, a constant function where* $c \in \{\wedge, \vee, \neg, \leftrightarrow\}$.
4. *If* $\varnothing \vdash_F v : \forall \alpha{:}k.\tau$*, then either* $v = \Lambda\alpha{:}k.e$ *or* $v = c$*, a polymorphic constant* $c \in \{\leq, =\}$.
5. *If* $\varnothing \vdash_w^F \tau : B$*, then* $\tau = \texttt{Bool}$ *or* $\tau = \texttt{Int}$.

Proof. Parts (1) - (4) are easily deduced from the λ_F typing rules in Figure 3.6 and the definition of $ty(c)$. Part (5) is clear from the well-formedness rules in Figure 3.5. □

We note that Lemma 4.3 is sufficient for our λ_{RF} metatheory. Our syntactic typing judgments in λ_{RF} respect those of λ_F. Specifically, if $\Gamma \vdash e : t$ and $\vdash_w \Gamma$, then $\lfloor\Gamma\rfloor \vdash_F e : \lfloor t \rfloor$. Therefore, we do not have to state and prove a separate Canonical Forms Lemma for λ_{RF}.

Lemma 4.4. *(Inversion of Typing)*

1. *If* $\Gamma \vdash_F c : \tau$*, then* $\tau = \lfloor \mathsf{ty}(c) \rfloor$.
2. *If* $\Gamma \vdash_F x : \tau$*, then* $x{:}\tau \in \Gamma$.
3. *If* $\Gamma \vdash_F e\ e_x : \tau$*, then there exists type* τ_x *such that* $\Gamma \vdash_F e : \tau_x \to \tau$ *and* $\Gamma \vdash_F e_x : \tau_x$.
4. *If* $\Gamma \vdash_F \lambda x.e : \tau$*, then* $\tau = \tau_x \to \tau'$ *and* $y{:}\tau_x, \Gamma \vdash_F e[y/x] : \tau'$ *for any* $y \notin \Gamma$ *and well-formed* τ_x.

5. *If* $\Gamma \vdash_F e[t] : \tau$*, then there exists type* σ *and kind* k *such that* $\Gamma \vdash_F e : \forall \alpha{:}k.\sigma$ *and* $\tau = \sigma[\lfloor t \rfloor/\alpha]$.

6. *If* $\Gamma \vdash_F \Lambda\alpha{:}k.e : \tau$*, then there exists type* τ' *and kind* k *such that* $\tau = \forall \alpha{:}k.\tau'$ *and* $\alpha' : k, \Gamma \vdash_F e[\alpha'/\alpha] : \tau'[\alpha'/\alpha]$ *for some* $\alpha' \notin \Gamma$.

7. *If* $\Gamma \vdash_F \texttt{let}\ x = e_x\ \texttt{in}\ e : \tau$*, then there exists type* τ_x *and* $y \notin \Gamma$ *such that* $\Gamma \vdash_F e_x : \tau_x$ *and* $y{:}\tau_x, \Gamma \vdash_F e[y/x] : \tau$.

8. *If* $\Gamma \vdash_F e : t : \tau$*, then* $\tau = \lfloor t \rfloor$ *and* $\Gamma \vdash_F e : \tau$.

9. *If* $\Gamma \vdash_F \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau$*, then* $\Gamma \vdash_F e : \texttt{Bool}$, $\Gamma \vdash_F e_1 : \tau$*, and* $\Gamma \vdash_F e_2 : \tau$.

Proof. This is clear from the definition of the typing rules for λ_F. Each premise can match only one rule because the λ_F rules are syntax directed. □

The Inversion of Typing Lemma does not hold in λ_{RF} due to the subtyping relation. For instance $x{:}\texttt{Int}\{\nu{:}\nu = 5\} \vdash x : \texttt{Int}$ but $x{:}\texttt{Int} \notin x{:}\texttt{Int}\{\nu{:}\nu = 5\}$. In Lemma 5.7 we state and prove an analogous result for λ_{RF} in the two cases needed to prove progress and preservation.

For each primitive constant or function, we need to know that the type $\lfloor \text{ty}(c) \rfloor$ relates to the λ_F type of $\delta(c, v)$ in the same manner as FT-APP.

Lemma 4.5. *(Primitives) For each built-in primitive* c*,*

1. *If* $\lfloor \text{ty}(c) \rfloor = \tau_x \to \tau$ *and* $\varnothing \vdash_F v_x : \tau_x$*, then* $\varnothing \vdash_F \delta(c, v_x) : \tau$.

2. *If* $\lfloor \text{ty}(c) \rfloor = \forall \alpha{:}k.\tau$ *and* $\varnothing \vdash_w \tau_\alpha : k$*, then* $\varnothing \vdash_F \delta_T(c, \tau_\alpha) : \tau[\tau_\alpha/\alpha]$.

Proof. 1. First consider $c \in \{\wedge, \vee, \neg\}$. Then $\lfloor \text{ty}(c) \rfloor = \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}$. Then by Lemma 4.3, $\varnothing \vdash_F v : \texttt{Bool}$ gives us that $v = \texttt{true}$ or $v = \texttt{false}$. For each possibility for c and v, we can build a judgment that $\varnothing \vdash_F \delta(c, v) : \texttt{Bool} \to \texttt{Bool}$. Similarly, if $c = \neg$ then $\lfloor \text{ty}(c) \rfloor = \texttt{Bool} \to \texttt{Bool}$ and $\delta(\neg, v) \in \{\texttt{true}, \texttt{false}\}$ can be typed at $\texttt{Bool}$. The analysis for the other monomorphic primitives is entirely similar.

2. Here c is the polymorphic $=$ and $\lfloor \mathsf{ty}(c) \rfloor = \forall \alpha{:}B.\, \alpha \to \alpha \to \texttt{Bool}$. By the Canonical Forms Lemma, $\tau = \texttt{Bool}$ or $\tau = \texttt{Int}$. In the former case, $\delta_T(c, \texttt{Bool}) = \leftrightarrow$, which we can type at $\texttt{Bool} \to \texttt{Bool} \to \texttt{Bool} = \lfloor \mathsf{ty}(c) \rfloor [\texttt{Bool}/\alpha]$. The case of `Int` is entirely similar because $\delta_T(c, \texttt{Int})$ is the monomorphic integer equality.

□

4.2.2 Preservation

The preservation lemma states that λ_F typing is preserved by evaluation.

Lemma 4.6. *(Preservation) If* $\varnothing \vdash_F e : \tau$ *and* $e \hookrightarrow e'$*, then* $\varnothing \vdash_F e' : \tau$.

Proof. We proceed by induction of the structure of $\varnothing \vdash_F e : \tau$. The cases of rules FT-PRIM, FT-VAR, FT-ABS, or FT-TABS cannot occur because e is a value and no value can take a step in our semantics. The interesting cases are for FT-APP and FT-TAPP. For applications of primitives, preservation requires the Primitives Lemma 4.5, while the general case needs a Substitution Lemma 4.7. We now give the full details of the other five cases:

Case FT-APP: We have $\varnothing \vdash_F e : \tau$ where $e \equiv e_1\ e_2$. Inverting, we have that there exists some type τ_2 such that $\varnothing \vdash_F e_1 : \tau_2 \to \tau$ and $\varnothing \vdash_F e_2 : \tau_2$. We split on five possible cases for the structure of e_1 and e_2. First, suppose $e_1 \equiv \lambda x.e_0$ and e_2 is a value. Then by rule E-APP and the determinism of our semantics, $e' \equiv e_0[e_2/x]$. By the Inversion of Typing (4.4), for some y we have $y{:}\tau_2 \vdash_F e_0[y/x] : \tau$. By the Substitution Lemma (4.7), substituting e_2 through for y gives us $\varnothing \vdash_F e_0[e_2/x] : \tau$ as desired because $e_0[y/x][e_2/y] = e_0[e_2/x]$. Second, suppose $e_1 \equiv \lambda x.e_0$ and e_2 is not a value. Then by the progress lemma (4.2), there exists a term e_2' such that $e_2 \hookrightarrow e_2'$. Then by rule E-PRAPP and the determinism of our semantics, $e' \equiv \lambda x.e_0\ e_2'$. Now, by the inductive hypothesis, $\varnothing \vdash_F e_2' : \tau_2$. Applying rule FT-APP, $\varnothing \vdash_F e_1\ e_2' : \tau$ as desired. Third, suppose $e_1 \equiv c$, a built-in primitive, and e_2 is a value. Then by rule E-PRIM and the determinism of the semantics, $e' \equiv \delta(c, e_2)$. By

the primitives lemma, $\varnothing \vdash_F \delta(c, e_2) : \tau$ as desired. Fourth, suppose $e_1 \equiv c$ and e_2 is not a value. Then we argue in the same manner as the second case. Finally, by the canonical forms lemma, e_1 cannot be any other value, so it must not be a value. Then by the progress lemma, there is a term e_1' such that $e_1 \hookrightarrow e_1'$. Then by rule E-PLAPP and the determinism of the semantics, $e' \equiv e_1'\ e_2$. By the inductive hypothesis, $\varnothing \vdash_F e_1' : \tau_2 \to \tau$. Applying rule FT-APP, $\varnothing \vdash_F e_1'\ e_2 : \tau$ as desired.

Case FT-TAPP: We have $\varnothing \vdash_F e : \tau$ where $e \equiv e_1[t]$ and $\tau \equiv \sigma[\lfloor t \rfloor/\alpha]$. Inverting (4.4), we have that $\varnothing \vdash_F e_1 : \forall\alpha{:}k.\sigma$ and $\varnothing \vdash_w^F \lfloor t \rfloor : k$. We split on three cases for the structure of e_1. First, suppose $e_1 \equiv \Lambda\alpha{:}k.e_0$. Then by rule E-TAPP and the determinism of the semantics, $e' \equiv e_0[t/\alpha]$. By the inversion of typing, for some α', we have $\alpha' : k \vdash_F e_0[\alpha'/\alpha] : \sigma[\alpha'/\alpha]$. By the Substitution Lemma (4.7), substituting $\lfloor t \rfloor$ through for α gives us $\varnothing \vdash_F e_0[t/\alpha] : \sigma[\lfloor t \rfloor/\alpha]$ as desired. Second, suppose $e_1 \equiv c$, a built-in primitive. Then by rule E-TPRIM and the determinism of the semantics, $e' \equiv \delta_T(c, \lfloor t \rfloor)$. By the primitives lemma, $\varnothing \vdash_F \delta_T(c, \lfloor t \rfloor) : \sigma[\lfloor t \rfloor/\alpha]$. Finally, by the canonical forms lemma, e_1 cannot be any other form of value, so it must not be a value. Then by the progress lemma, there is a term e_1' such that $e_1 \hookrightarrow e_1'$. Then by rule E-PTAPP and the deterministic semantics $e' \equiv e_1'[t]$. By the inductive hypothesis, $\varnothing \vdash_F e_1' : \forall\alpha{:}k.\sigma$. Applying rule FT-TAPP, $\varnothing \vdash_F e_1'[t] : \sigma[\lfloor t \rfloor/\alpha]$ as desired.

Case FT-LET: We have $\varnothing \vdash_F e : \tau$ where $e \equiv \texttt{let}\ x = e_1\ \texttt{in}\ e_2$. Inverting, we have that $y : \tau_1 \vdash_F e_2[y/x] : \tau$ and $\varnothing \vdash_F e_1 : \tau_1$ for some type τ_1. By the progress lemma either e_1 is a value or there is a term e_1' such that $e_1 \hookrightarrow e_1'$. In the former case, rule E-LET and determinism give us $e' \equiv e_2[e_1/x]$. By the Substitution Lemma (substituting e_1 for x), we have $\varnothing \vdash_F e_2[e_1/x] : \tau$ as desired because $e_2[e_1/x] = e_2[y/x][e_1/y]$. In the latter case, by rule E-PLET and determinism give us, $e' \equiv \texttt{let}\ x = e_1'\ \texttt{in}\ e_2$. By the inductive hypothesis we have that $\varnothing \vdash_F e_1' : \tau_1$ and by rule FT-LET we have $\varnothing \vdash_F \texttt{let}\ x = e_1'\ \texttt{in}\ e_2 : \tau$.

Case FT-ANN: We have $\varnothing \vdash_F e : \tau$ where $e \equiv e_1 : t$. Inverting, we have the $\varnothing \vdash_F e_1 : \tau$ and $\tau = \lfloor t \rfloor$. By the progress lemma, either e_1 is a value or there is a term e_1' such that $e_1 \hookrightarrow e_1'$. In the former case, by rule E-ANN and the determinism of the semantics, $e' \equiv e_1$. Then we already have that $\varnothing \vdash_F e' : \tau$ In the latter case, rule E-PANN and determinism give us $e' \equiv e_1' : t$. By the inductive hypothesis we have that $\varnothing \vdash_F e_1' : \tau$. By rule FT-ANN we conclude $\varnothing \vdash_F e_1' : t : \tau$.

Case FT-IF: We have $\varnothing \vdash_F e : \tau$ where $e \equiv \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. Inverting, we have that $\varnothing \vdash_F e_0 : \texttt{Bool}$ and $\varnothing \vdash_F e_1 : \tau$, and $\varnothing \vdash_F e_2 : \tau$. By the progress lemma either e_0 is a value or there is a term e_0' such that $e_0 \hookrightarrow e_0'$. In the former case, the Canonical Forms Lemma (4.3) tells us that $e_0 = \texttt{true}$ or $\texttt{false}$. By determinism of the semantics we have that $e' \equiv e_1$ or $e' \equiv e_2$ respectively. In either case we can immediately conclude that e' has the desired type. In the latter case above, rule E-PIF and determinism give us, $e' \equiv \texttt{if}\ e_0'\ \texttt{then}\ e_1'\ \texttt{else}\ e_2$. By the inductive hypothesis we have that $\varnothing \vdash_F e_0' : \texttt{Bool}$ and by rule FT-IF we have $\varnothing \vdash_F \texttt{if}\ e_0'\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau$.

□

The proof of preservation for λ_{RF} differs in two cases above. In T-APP and T-TAPP, we must use the Inversion of Typing lemma (5.7) from λ_{RF} because the presence of rule T-SUB prevents us from inferring the last rule used to type a term or type abstraction. Furthermore, in case T-APP the substitution lemma would give us that $\varnothing \vdash e' : t[v_x/x]$ for some value v_x. However, we need to show preservation of the existential type $\exists x{:}t_x.t$. This is done by using rule S-WIT to show that, in fact, $\varnothing \vdash t[v_x/x] \preceq \exists x{:}t_x.t$.

Substitution Lemma To prove type preservation when a lambda or type abstraction is applied, we proved that the substituted result has the same type, as established by the Substitution Lemma:

Lemma 4.7. *(Substitution) If* $\Gamma \vdash_F v_x : \tau_x$ *and* $\Gamma \vdash_w^F \lfloor t_\alpha \rfloor : k_\alpha$*, then*

1. *if* $\Gamma', x{:}\tau_x, \Gamma \vdash_F e : \tau$ *and* $\vdash_w \Gamma$, *then* $\Gamma', \Gamma \vdash_F e[v_x/x] : \tau$ *and*

2. *if* $\Gamma', \alpha{:}k_\alpha, \Gamma \vdash_F e : \tau$ *and* $\vdash_w \Gamma$, *then* $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F e[t_\alpha/\alpha] : \tau[\lfloor t_\alpha \rfloor/\alpha]$.

Proof. We give the proofs for part (2); part (1) is similar but slightly simpler because term variables do not appear in types in λ_F We proceed by induction on the derivation tree of the typing judgment $\Gamma', \alpha{:}k_\alpha, \Gamma \vdash_F e : \tau$.

Case FT-PRIM: We have $e \equiv c$ and $\Gamma', \alpha{:}k_\alpha, \Gamma \vdash_F c : \lfloor \mathrm{ty}(c) \rfloor$. Neither c nor $\mathrm{ty}(c)$ has any free variables, so each is unchanged under substitution. Then by rule T-PRIM we conclude $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F c : \lfloor \mathrm{ty}(c) \rfloor$ because the environment may be chosen arbitrarily.

Case FT-VAR: We have $e \equiv x$; by inversion, we get that $x{:}\tau \in \Gamma', \alpha{:}k_\alpha, \Gamma$. We must have $\alpha \neq x$, so there are two cases to consider for where x can appear in the environment. If $x{:}\tau \in \Gamma$, then τ cannot contain α as a free variable because x is bound first in the environment (which grows from right to left). Then $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F x : \tau$ as desired because $\tau[\lfloor t_\alpha \rfloor/\alpha] = \tau$. Otherwise, $x{:}\tau \in \Gamma'$ and so $x{:}\tau[\lfloor t_\alpha \rfloor/\alpha] \in \Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma$. Thus $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F x : \tau[\lfloor t_\alpha \rfloor/\alpha]$. (In part (1), we have an additional case where $\Gamma', x{:}\tau, \Gamma \vdash_F x : \tau$. We have $x[v_x/x] = v_x$, and so we can apply the Weakening Lemma (4.8) to $\Gamma \vdash_F v_x : \tau$ to obtain $\Gamma', \Gamma \vdash_F v_x : \tau$.)

Case FT-APP: We have $e \equiv e_1\ e_2$. By inversion, we have that $\Gamma', \alpha{:}k_\alpha, \Gamma \vdash_F e_1 : \tau_x \to \tau$ and $\Gamma', \alpha{:}k_\alpha, \Gamma \vdash_F e_2 : \tau_x$. Applying the inductive hypothesis to both of these, we get $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F e_1[t_\alpha/\alpha] : \tau_x \to \tau[\lfloor t_\alpha \rfloor/\alpha]$ and $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F e_2[t_\alpha/\alpha] : \tau_x[\lfloor t_\alpha \rfloor/\alpha]$. Combining these by rule FT-APP, we conclude $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F e_1\ e_2[t_\alpha/\alpha] : \tau[\lfloor t_\alpha \rfloor/\alpha]$.

Case FT-ABS: We have $e \equiv \lambda x.e_1$ and $\tau \equiv \tau_x \to \tau_1$. By inversion we have that for any fresh y, both $y{:}\tau_x, \Gamma', \alpha{:}k_\alpha, \Gamma \vdash_F e_1[y/x] : \tau_1$ and $\Gamma', \alpha{:}k_\alpha, \Gamma \vdash^F_w \tau_x : k_x$. By the inductive hypothesis, and the Substitution Lemma for well-formedness judgments, we have $y{:}\tau_x[\lfloor t_\alpha \rfloor/\alpha], \Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F e_1[t_\alpha/\alpha][y/x] : \tau_1[\lfloor t_\alpha \rfloor/\alpha]$ and $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash^F_w \tau_x[\lfloor t_\alpha \rfloor/\alpha] : k_x$, where we can switch the order of substitutions because y does not appear free in the well-formed type t_α. Then we can conclude by applying rule FT-ABS that $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F \lambda x.e_1[t_\alpha/\alpha] : \tau_x \to \tau_1[\lfloor t_\alpha \rfloor/\alpha]$.

Case FT-TAPP: We have $e \equiv e'\ [t']$ and $\tau \equiv \tau'[\lfloor t' \rfloor/\alpha']$. By inversion, $\Gamma', \alpha{:}k_\alpha, \Gamma \vdash_F e' : \forall\, \alpha'{:}k'.\, \tau'$ and $\Gamma', \alpha{:}k_\alpha, \Gamma \vdash^F_w \lfloor t' \rfloor : k'$. By the inductive hypothesis, $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F e'[t_\alpha/\alpha] : \forall\, \alpha'{:}k'.\, \tau'[\lfloor t_\alpha \rfloor/\alpha]$ and $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash^F_w \lfloor t' \rfloor[\lfloor t_\alpha \rfloor/\alpha] : k'$. By the definition of refinement erasure, $\lfloor t \rfloor[\lfloor t_\alpha \rfloor/\alpha'] = \lfloor t'[t_\alpha/\alpha] \rfloor$. By applying rule FT-TAPP, we get $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F e'[t_\alpha/\alpha][t'[t_\alpha/\alpha]] : \tau'[\lfloor t_\alpha \rfloor/\alpha][\lfloor t'[t_\alpha/\alpha] \rfloor/\alpha']$. By the definition of substitution we have $e'[t_\alpha/\alpha][t'[t_\alpha/\alpha]] = e'[t'][t_\alpha/\alpha]$ and by the commutativity rules for substitution, $\tau'[\lfloor t_\alpha \rfloor/\alpha][\lfloor t'[t_\alpha/\alpha] \rfloor/\alpha'] = \tau'[\lfloor t' \rfloor/\alpha'][\lfloor t_\alpha \rfloor/\alpha]$. Therefore, we conclude that $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F e[t_\alpha/\alpha] : \tau[\lfloor t_\alpha \rfloor/\alpha]$.

Case FT-TABS: We have $\Gamma', \alpha{:}k_\alpha, \Gamma \vdash_F e : \tau$ where $e \equiv \Lambda\alpha_0{:}k'.e'$ and $\tau \equiv \forall\, \alpha_0{:}k'.\, \tau'$. By inversion, for any fresh type variable α', $\alpha'{:}k', \Gamma', \alpha{:}k_\alpha, \Gamma \vdash_F e'[\alpha'/\alpha_0] : \tau'[\alpha'/\alpha_0]$ and $\alpha'{:}k', \Gamma', \alpha{:}k_\alpha, \Gamma \vdash^F_w \tau'[\alpha'/\alpha_0] : k'$. By the inductive hypothesis

$$\alpha'{:}k', \Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F e'[\alpha'/\alpha_0][t_\alpha/\alpha] : \tau'[\alpha'/\alpha_0][\lfloor t_\alpha \rfloor/\alpha]$$

and by the Substitution Lemma for λ_F well-formedness judgments

$$\alpha'{:}k', \Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash^F_w \tau'[\alpha'/\alpha_0][\lfloor t_\alpha \rfloor/\alpha] : k'.$$

Because α' is chosen to be fresh, we must have $\alpha \neq \alpha'$ and $\alpha_0 \neq \alpha'$. Moreover, $\lfloor t_\alpha \rfloor$ contains only free variables from Γ, so $e'[\alpha'/\alpha_0][t_\alpha/\alpha] = e'[t_\alpha/\alpha][\alpha'/\alpha_0]$ and $\tau'[\alpha'/\alpha_0][\lfloor t_\alpha \rfloor/\alpha] = \tau'[\lfloor t_\alpha \rfloor/\alpha][\alpha'/\alpha_0]$. Then by rule FT-TABS,

$$\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F \Lambda\alpha_0{:}k'.(e'[t_\alpha/\alpha]) : \forall\, \alpha_0{:}k'.(\tau'[\lfloor t_\alpha \rfloor/\alpha]).$$

By definition of substitution, we can rewrite the above as

$$\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F (\Lambda\alpha_0{:}k'.e')[t_\alpha/\alpha] : (\forall\, \alpha_0{:}k'.\, \tau')[\lfloor t_\alpha \rfloor/\alpha]$$

as desired.

Case FT-LET: We have $e \equiv \texttt{let}\ x = e_1\ \texttt{in}\ e_2$ and by inversion we have that for some type τ_1, $\Gamma', \alpha{:}k_\alpha, \Gamma \vdash_F e_1 : \tau_1$ and for some $y \notin \Gamma', \alpha{:}k_\alpha, \Gamma$, $y{:}\tau_1, \Gamma', \alpha{:}k_\alpha, \Gamma \vdash_F e_2[y/x] : \tau$. By the inductive hypothesis, we have that $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F e_1[t_\alpha/\alpha] : \tau_1[\lfloor t_\alpha \rfloor/\alpha]$ and $y{:}\tau_1[t_\alpha/\alpha], \Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F e_2[t_\alpha/x] : \tau[\lfloor t_\alpha \rfloor/\alpha]$. Then by rule FT-LET we conclude $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F \texttt{let}\ x = e_1[t_\alpha/\alpha]\ \texttt{in}\ e_2[t_\alpha/\alpha] : \tau[\lfloor t_\alpha \rfloor/\alpha]$.

Case FT-ANN: We have $e \equiv e' : t$ and by inversion we have that $\lfloor t \rfloor = \tau$ and $\Gamma', \alpha{:}k_\alpha, \Gamma \vdash_F e' : \tau$. By the inductive hypothesis, we have $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F e'[t_\alpha/\alpha] : \tau[\lfloor t_\alpha \rfloor/\alpha]$. By our definition of refinement erasure, we have $\lfloor t[t_\alpha/\alpha] \rfloor = \lfloor t \rfloor[\lfloor t_\alpha \rfloor/\alpha]$, and we have $e' : t[t_\alpha/\alpha] = e'[t_\alpha/\alpha] : t[t_\alpha/\alpha]$. Thus by rule FT-ANN, $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F e' : t[t_\alpha/\alpha] : \lfloor t \rfloor[\lfloor t_\alpha \rfloor/\alpha]$.

Case FT-IF: We have $e \equiv \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ and by inversion we have that $\Gamma', \alpha{:}k_\alpha, \Gamma \vdash_F e_0 : \texttt{Bool}$ and $\Gamma', \alpha{:}k_\alpha, \Gamma \vdash_F e_1 : \tau$ and $\Gamma', \alpha{:}k_\alpha, \Gamma \vdash_F e_2 : \tau$. By the inductive hypothesis, we have $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F e_0[t_\alpha/\alpha] : \texttt{Bool}$ and $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F e_i[t_\alpha/\alpha] : \tau[\lfloor t_\alpha \rfloor/\alpha]$ for each $i = 1, 2$. Then by rule FT-IF we have

$$\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F \texttt{if}\ e_0[t_\alpha/\alpha]\ \texttt{then}\ e_1[t_\alpha/\alpha]\ \texttt{else}\ e_2[t_\alpha/\alpha] : \tau[\lfloor t_\alpha \rfloor/\alpha]$$

By the definition of substitution, we have the equality $\texttt{if}\ e_0[t_\alpha/\alpha]\ \texttt{then}\ e_1[t_\alpha/\alpha]\ \texttt{else}\ e_2[t_\alpha/\alpha] = (\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2)[t_\alpha/\alpha]$, which gives us the desired judgment. □

Because we encoded our typing rules using cofinite quantification the proof above does not require a renaming lemma, but the rules that lookup environments (rules T-VAR and WF-VAR) do need a *Weakening Lemma*:

Lemma 4.8. *(Weakening) If* $\Gamma_1, \Gamma_2 \vdash_F e : \tau$ *and* $x, \alpha \notin \Gamma_1, \Gamma_2$*, then*

1. $\Gamma_1, x{:}\tau_x, \Gamma_2 \vdash_F e : \tau$*, and*
2. $\Gamma_1, \alpha{:}k, \Gamma_2 \vdash_F e : \tau$.

The proof is fairly similar to the proof of the Substitution Lemma above, and we omit it here.

Acknowledgements for Chapter 4

This chapter is adapted from unpublished material that was originally prepared (as an appendix) for "Mechanizing Refinement Types" in the proceedings of the 51^{st} ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2024), by Michael Borkowski, Niki Vazou, and Ranjit Jhala. The dissertation author was the primary investigator and author of this material.

Chapter 5

Soundness of λ_{RF}

Our development of the metatheory for λ_F (§ 4.2) followed the standard presentation of System F's metatheory by Pierce [38]. The main difference is that ours includes well-formedness of types and environments, which help with mechanization [43] and are crucial in λ_{RF} when formalizing refinements.

Figure 5.1 charts the overall landscape of our formal development as a dependency graph of the main lemmas which establish meta-theoretic properties of the different judgments for λ_F and λ_{RF}. Nodes shaded light grey represent lemmas in the metatheories for both λ_F and λ_{RF}. The dark grey nodes denote lemmas that only appear in λ_{RF}. An arrow shows a dependency: the lemma at the *tail* is used in the proof of the lemma at the *head*. Solid arrows are dependencies in λ_{RF} only. The chart already shows that the metatheory of the refined calculus λ_{RF} is much more complex that the one of the unrefined system λ_F, as also shown by the summary of our mechanization (Table 7.1).

In the rest of this chapter we establish denotational soundness (§ 5.1) and type safety (§ 5.2) and we flesh out the skeleton of Figure 5.1, *i.e.* the inversion (§ 5.3), substitution (§ 5.4), and narrowing (§ 5.5) lemmas.

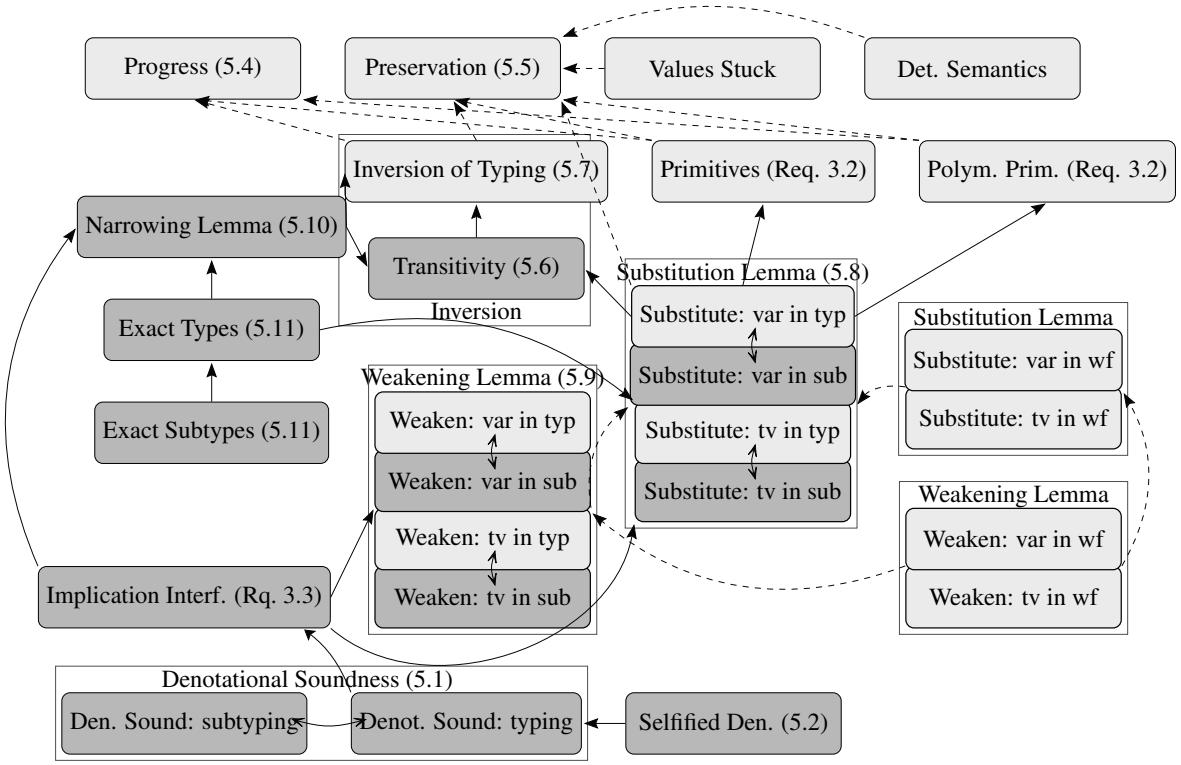

Figure 5.1: Dependencies in the metatheory. We write "var" and "tv" to resp. abbreviate term and type variables.

5.1 Denotational Soundness

Denotational soundness connects syntactic typing and subtyping with the type denotations (of Figure 3.8). For typing, it states that if $\Gamma \vdash e : t$, then when e is closed by any closing substitution of Γ it evaluates to a value that belongs in the denotation of the closed t. For subtyping, if $\Gamma \vdash s \preceq t$, then under all closing substitutions, the denotation of the former type is contained in the latter:

Theorem 5.1. *(Denotational Soundness)*

1. *If* $\Gamma \vdash e : t$ *and* $\vdash_w \Gamma$ *and* $\theta \in [\![\Gamma]\!]$ *then* $\theta(e) \hookrightarrow^* v \in [\![\theta(t)]\!]$ *for some value* v.

2. *If* $\Gamma \vdash t_1 \preceq t_2$ *and* $\vdash_w \Gamma$ *and* $\Gamma \vdash_w t_1 : k_1$ *and* $\Gamma \vdash_w t_2 : k_2$ *and* $\theta \in [\![\Gamma]\!]$ *then* $[\![\theta(t_1)]\!] \subseteq [\![\theta(t_2)]\!]$.

The proof is by mutual induction on the structure of the judgments $\Gamma \vdash e : t$ and $\Gamma \vdash t_1 \preceq t_2$ respectively. Our rule T-VAR mentions selfification, so we use Lemma 5.2 for that case.

Lemma 5.2. *(Selfified Denotations) If* $\varnothing \vdash_w t : k$, $\varnothing \vdash e : t$, $e \hookrightarrow^* v$ *for some* $v \in [\![t]\!]$ *then* $v \in [\![\mathsf{self}(t, e, k)]\!]$.

This lemma captures the intuition that if $v \in [\![b\{x : p\}]\!]$ (*i.e.* if v has base type b and $p[v/x] \hookrightarrow^* \texttt{true}$), then we have $v \in [\![b\{x : p \wedge x = v\}]\!]$ as $(p \wedge x = v)[v/x]$ certainly evaluates to `true`. The full proof also handles the case that t is an existential type as well as selfification by an expression e that evaluates to v.

5.2 Type Safety

The type safety theorem states that a well-typed term does not get stuck: *i.e.* either evaluates to a value or can step to another term (progress) of the same type (preservation).

Theorem 5.3. *(Type Safety of* λ_{RF}*)*

1. *(Type Safety) If* $\varnothing \vdash e : t$ *and* $e \hookrightarrow^* e'$, *then* e' *is a value or* $e' \hookrightarrow e''$ *for some* e''.
2. *(No Error) If* $\varnothing \vdash e : t$ *and* $e \hookrightarrow^* e'$, *then* $e' \neq \texttt{error}$.

The No Error property explicitly states that well-typed terms cannot evaluate to the term `error` (that encodes stuck terms) and is a direct implication of type safety. We prove type safety by induction on the length of the sequence of steps $e \hookrightarrow^* e'$, using preservation and progress.

Progress The progress lemma says a well-typed term is a value or steps to some other term.

Lemma 5.4. *(Progress) If* $\varnothing \vdash e : t$, *then* e *is a value or* $e \hookrightarrow e'$ *for some* e'.

The proof is by induction on the typing derivation using the primitives Requirement 3.2, that we proved for our built-in primitives, and the inversion of typing lemma.

Preservation The preservation lemma states that typing is preserved by evaluation.

Lemma 5.5. *(Preservation) If* $\varnothing \vdash e : t$ *and* $e \hookrightarrow e'$*, then* $\varnothing \vdash e' : t$*.*

The proof is by structural induction on the derivation of the typing judgment and implicitly uses the inversion lemma. We use the determinism of the operational semantics (lemma 3.1) and the canonical forms lemma to case split on e to determine e'. The interesting cases are for FT-APP and FT-TAPP that require a substitution Lemma 5.8. Next, let's see the three main lemmas used in the preservation and progress proofs.

5.3 Inversion of Typing Judgments

The region of Figure 5.1 labelled "Inversion" accounts for the fact that, due to subtyping chains, the typing judgment in λ_{RF} is not syntax-directed. First, we establish that subtyping is transitive:

Lemma 5.6. *(Transitivity) If* $\Gamma \vdash_w t_1 : k_1$*,* $\Gamma \vdash_w t_3 : k_3$*,* $\vdash_w \Gamma$*,* $\Gamma \vdash t_1 \preceq t_2$*,* $\Gamma \vdash t_2 \preceq t_3$*, then* $\Gamma \vdash t_1 \preceq t_3$*.*

The proof consists of a case-split on the possible rules for $\Gamma \vdash t_1 \preceq t_2$ and $\Gamma \vdash t_2 \preceq t_3$. When the last rule used in the former is S-WIT and the latter is S-BIND, we require the substitution Lemma 5.8. As Aydemir et al. [2], we use the narrowing Lemma 5.10 for the transitivity for function types.

Inverting Typing Judgments We use the transitivity of subtyping to prove some non-trivial lemmas that let us "invert" the typing judgments to recover information about the underlying terms and types. We describe the non-trivial case which pertains to type and value abstractions:

Lemma 5.7. *(Inversion of* T-ABS*,* T-TABS*)*

1. *If* $\Gamma \vdash (\lambda w.e) : x{:}t_x \to t$ *and* $\vdash_w \Gamma$*, then for all* $y \notin \Gamma$*,* $y{:}t_x, \Gamma \vdash e[y/w] : t[y/x]$*.*

2. *If* $\Gamma \vdash (\Lambda\alpha_1{:}k_1.e) : \forall\,\alpha{:}k.t$ *and* $\vdash_w \Gamma$, *then for all* $\alpha' \notin \Gamma$, $\alpha'{:}k, \Gamma \vdash e[\alpha'/\alpha_1] : t[\alpha'/\alpha]$.

If $\Gamma \vdash (\lambda w.e) : x{:}t_x \to t$, then we cannot directly invert the typing judgment to get a judgment for the body e of $\lambda w.e$. Perhaps the last rule used was T-SUB, and inversion only tells us that there exists a type t_1 such that $\Gamma \vdash (\lambda w.e) : t_1$ and $\Gamma \vdash t_1 \preceq x{:}t_x \to t$. Inverting again, we may in fact find a chain of types $t_{i+1} \preceq t_i \preceq \cdots \preceq t_2 \preceq t_1$ which can be arbitrarily long. But the proof tree must be finite so eventually we find a type $w{:}s_w \to s$ such that $\Gamma \vdash (\lambda w.e) : w{:}s_w \to s$ and $\Gamma \vdash w{:}s_w \to s \preceq x{:}t_x \to t$ (by transitivity) and the last rule was T-ABS. Then inversion gives us that for any $y \notin \Gamma$ we have $y{:}s_w, \Gamma \vdash e : s[y/w]$. To get the desired typing judgment, we must use the narrowing Lemma 5.10 to obtain $y{:}t_x, \Gamma \vdash e : s[y/w]$. Finally, we use T-SUB to derive $y{:}t_x, \Gamma \vdash e : t[y/w]$.

5.4 Substitution Lemma

In λ_{RF}, unlike unrefined calculi such as λ_F, typing and subtyping are mutual dependent. Due to this dependency, both the substitution the weakening lemmas must now be proven in a mutually recursive form:

Lemma 5.8. *(Substitution)*

1. *If* $\Gamma_1, x{:}t_x, \Gamma_2 \vdash s \preceq t$, $\vdash_w \Gamma_2$, *and* $\Gamma_2 \vdash v_x : t_x$, *then* $\Gamma_1[v_x/x], \Gamma_2 \vdash s[v_x/x] \preceq t[v_x/x]$.

2. *If* $\Gamma_1, x{:}t_x, \Gamma_2 \vdash e : t$, $\vdash_w \Gamma_2$, *and* $\Gamma_2 \vdash v_x : t_x$, *then* $\Gamma_1[v_x/x], \Gamma_2 \vdash e[v_x/x] : t[v_x/x]$.

3. *If* $\Gamma_1, \alpha{:}k, \Gamma_2 \vdash s \preceq t$, $\vdash_w \Gamma_2$, *and* $\Gamma_2 \vdash_w t_\alpha : k$, *then* $\Gamma_1[t_\alpha/\alpha], \Gamma_2 \vdash s[t_\alpha/\alpha] \preceq t[t_\alpha/\alpha]$.

4. *If* $\Gamma_1, \alpha{:}k, \Gamma_2 \vdash e : t$, $\vdash_w \Gamma_2$, *and* $\Gamma_2 \vdash_w t_\alpha : k$, *then* $\Gamma_1[t_\alpha/\alpha], \Gamma_2 \vdash e[t_\alpha/\alpha] : t[t_\alpha/\alpha]$.

The proof goes by induction on the derivation trees. The main difficulty arises in substituting some type t_α for variable α in $\Gamma_1, \alpha{:}k, \Gamma_2 \vdash \alpha\{x_1 : p\} \preceq \alpha\{x_2 : q\}$ because t_α must be strengthened by the refinements p and q respectively. Because we encoded our typing rules using

cofinite quantification [3] the proof does not require a renaming lemma, but the rules that lookup environments (rules T-VAR and WF-VAR) do need a *weakening Lemma*:

Lemma 5.9. *(Weakening) If* $x, \alpha \notin \Gamma_1, \Gamma_2$*, then*

1. *if* $\Gamma_1, \Gamma_2 \vdash e : t$ *then* $\Gamma_1, x{:}t_x, \Gamma_2 \vdash e : t$ *and* $\Gamma_1, \alpha{:}k, \Gamma_2 \vdash e : t$.
2. *if* $\Gamma_1, \Gamma_2 \vdash s \preceq t$ *then* $\Gamma_1, x{:}t_x, \Gamma_2 \vdash s \preceq t$ *and* $\Gamma_1, \alpha{:}k, \Gamma_2 \vdash s \preceq t$.

The proof is by mutual induction on the derivation of the typing and subtyping judgments.

5.5 Narrowing

The narrowing lemma says that whenever we have a judgment where a binding $x{:}t_x$ appears in the binding environment, we can replace t_x by any subtype s_x. The intuition here is that the judgment holds under the replacement because we are making the context more specific.

Lemma 5.10. *(Narrowing) If* $\Gamma_2 \vdash s_x <: t_x$*,* $\Gamma_2 \vdash_w s_x : k_x$*, and* $\vdash_w \Gamma_2$ *then*

1. *if* $\Gamma_1, x{:}t_x, \Gamma_2 \vdash_w t : k$*, then* $\Gamma_1, x{:}s_x, \Gamma_2 \vdash_w t : k$.
2. *if* $\Gamma_1, x{:}t_x, \Gamma_2 \vdash t_1 <: t_2$*, then* $\Gamma_1, x{:}s_x, \Gamma_2 \vdash t_1 <: t_2$.
3. *if* $\Gamma_1, x{:}t_x, \Gamma_2 \vdash e : t$*, then* $\Gamma_1, x{:}s_x, \Gamma_2 \vdash e : t$.

The narrowing proof requires an exact typing Lemma 5.11 which says that a subtyping judgment $\Gamma \vdash s \preceq t$ is preserved after selfification on both types. Similarly, whenever we can type a value v at type t then we also type v at the type t selfified by v.

Lemma 5.11. *(Exact Typing)*

1. *If* $\Gamma \vdash e : t$*,* $\vdash_w \Gamma$*,* $\Gamma \vdash_w t : k$*, and* $\Gamma \vdash s \preceq t$*, then* $\Gamma \vdash \mathrm{self}(s, v, k) \preceq \mathrm{self}(t, v, k)$.
2. *If* $\Gamma \vdash v : t$*,* $\vdash_w \Gamma$*, and* $\Gamma \vdash_w t : k$*, then* $\Gamma \vdash v : \mathrm{self}(t, v, k)$.

Acknowledgements for Chapter 5

This chapter is adapted from "Mechanizing Refinement Types" in the proceedings of the 51st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2024), by Michael Borkowski, Niki Vazou, and Ranjit Jhala. The dissertation author was the primary investigator and author of this material.

Chapter 6

LIQUIDHASKELL & Refined Data Propositions

In Chapter 7 we will present how we proved λ_{RF} soundness in LIQUIDHASKELL. To do so, we developed *refined data propositions*, a novel feature of LIQUIDHASKELL that made such a meta-theoretic proof possible. Although LIQUIDHASKELL had been previously used to prove theorems inductively about provably terminating user defined functions, these extensions were needed to reason about potentially non-terminating properties, such as the typing judgment of λ_{RF}.

6.1 LIQUIDHASKELL

LIQUIDHASKELL's core proof system is λ_{RF}, that is, it is using the typing judgment presented in Figure 3.6 to check whether a Haskell program satisfies its refinement type annotations. The expression language checked by LIQUIDHASKELL is GHC's intermediate language (CORESYN [47]) which is a superset of λ_{RF} that also includes literals, datatypes, and coercions. Thus, LIQUIDHASKELL's typing judgment is extended to include these constructs. To guess the

unknown types of Figure 3.6 (*i.e.* in the rules T-SUB and T-LET) and make the typing judgment algorithmic, LIQUIDHASKELL implements the refinement type inference algorithm of liquid types [42]. To check the implications LIQUIDHASKELL uses the I-LOG rule of § 3.3.4.2 which are automatically discharged by an SMT solver.

LIQUIDHASKELL as a Theorem Prover Equipped with the SMT solver, LIQUIDHASKELL can be used to prove theorems over theories known to the SMT solver. For example, that addition over integers is associative and that for every integer there exists a larger one, as encoded by the below functions:

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


exLg :: x:Int → (y::Int,{v:() | y > x })
exLg x = (x+1, ())
```

These definitions use lambda abstraction and dependent pairs to respectively encode the universal and existential quantifiers. To encode logical terms, such as `y > x`, they refine the unit type with such terms. Building upon this idea, LIQUIDHASKELL has been extensively used to prove theorems, using recursive Haskell definitions to encode inductive proofs and refinement reflection [53] to allow user-defined terminating functions into the refinement logic. Yet, the proving power of LIQUIDHASKELL was limited because only provably terminating functions can be used in the refinement logic and the proofs were implicitly performed by the SMT solver. Thus, the programmer could not inspect the proof terms.

6.2 Refined Data Propositions

Refined data propositions encode COQ-style inductive predicates in LIQUIDHASKELL by refining Haskell's data types, allowing the programmer to write plain Haskell functions to provide constructive proofs for user-defined propositions. Here, for exposition, we present the four steps we followed in the mechanization of λ_{RF} to define the "has-type" proposition and then use it to type the primitive one.

Step 1: Reifying Propositions as Data Our first step is to represent the propositions of interest as plain Haskell data. For example, we can define the following types (suffixed `Pr` for "proposition"):

```
data HasTyPr   = HasTyPr    Env  Expr Type
data IsSubTyPr = IsSubTyPr  Env  Type Type
```

Thus, `HasTyPr Γ e t` and `IsSubTyPr Γ s t` respectively represent the *propositions* $\Gamma \vdash e : t$ and $\Gamma \vdash s \preceq t$, which say that `e` can be typed as `t` under environment Γ and that `t` is a subtype of `t'` under Γ.

Step 2: Reifying Evidence as Data Next, we reify evidence, *i.e. derivation trees* as data by defining Haskell data types with a *single constructor per derivation rule*. For example, we define the data type `HasTyEv` to encode the typing rules of Figure 3.6, with constructors that match the names of each rule.

```
data HasTyEv where
  TPrim :: Env → Prim → HasTyEv
  TSub  :: Env → Expr → Type → Type → HasTyEv → IsSubTyEv → HasTyEv
  ...
```

Using these data one can construct derivation trees. For instance, `TPrim Empty (PInt 1)::HasTyEv` is the tree that types the primitive one under the empty environment.

Step 3: Relating Evidence to its Propositions Next, we specify the relationship between

the evidence and the proposition that it establishes, via a refinement-level *uninterpreted function*:

```
measure hasTyEvPr   :: HasTyEv → HasTyPr
measure isSubTyEvPr :: IsSubTyEv → IsSubTyPr
```

The above signatures declare that `hasTyEvPr` (resp. `isSubTyEvPr`) is a refinement-level function that maps has-type (resp. is-subtype) evidence to its corresponding proposition. We can now use these uninterpreted functions to define *type aliases* that denote well-formed evidence that establishes a proposition. For example, consider the (refined) type aliases

```
type HasTy   Γ e t = {ev:HasTyEv   | hasTyEvPr ev == HasTyPr Γ e t }
type IsSubTy Γ s t = {ev:IsSubTyEv | isSubTyEvPr ev == IsSubTyPr Γ s t }
```

The definition stipulates that the type `HasTy Γ e t` is inhabited by evidence (of type `HasTyEv`) that establishes the typing proposition `HasTyPr Γ e t`. Similarly, `IsSubTy Γ s t` is inhabited by evidence (of type `IsSubTyEv`) that establishes the subtyping proposition `IsSubTyPr Γ s t`. Note that the first three steps have only defined separate data types for propositions and evidence, and *specified* the relationship between them via uninterpreted functions in the refinement logic.

Step 4: Refining Evidence to Establish Propositions Finally, we *implement* the relationship between evidence and propositions *refining* the types of the evidence data constructors (rules) with pre-conditions that require the rules' premises and post-conditions that ensure the rules' conclusions. For example, we connect the evidence and proposition for the typing relation by refining the data constructors for `HasTyEv` using their respecting typing rule from Figure 3.6.

```
data HasTyEv where
  TPrim :: Γ:Env → c:Prim → HasTy Γ (Prim c) (ty c)
  TSub  :: Γ:Env → e:Expr → s:Type → t:Type
        → HasTy Γ e s → IsSubTy Γ s t → HasTy Γ e t
  ...
```

The constructors `TPrim` and `TSub` respectively encode the rules T-PRIM and T-SUB (with well-formedness elided for simplicity). The refinements on the input types, which encode the premises

of the rules, are checked whenever these constructors are used. The refinement on the output type (being evidence of a specific proposition) is axiomatized to encode the conclusion of the rules. For example, the type for `TSub` says that "for all Γ, e, s, t, given evidence that $\Gamma \vdash e : s$ and $\Gamma \vdash s \preceq t$", the constructor returns "evidence that $\Gamma \vdash e : t$".

Programs as Constructive Proofs Thus, the constructor refinements crucially ensure that only well-formed pieces of evidence can be constructed, and simultaneously, precisely track the proposition established by the evidence. This lets the programmer write plain Haskell terms as constructive proofs, and LIQUIDHASKELL ensures that those terms indeed establish the proposition stipulated by their type. For example, the below Haskell term is proof that the literal `1` has the type $\mathsf{Int}\{\mathsf{v} : \mathsf{v} = 1\}$

```
oneTy :: HasTy Empty (EPrim (PInt 1)) {v:Int | v == 1}
oneTy = TPrim Empty (PInt 1)
```

If instead, the programmer wrote `oneTy = TPrim Empty (PInt 2)`, LIQUIDHASKELL would reject this as the modified evidence does not establish the proposition described in the type.

Implementation of Data Propositions Data propositions are a novel feature required to encode inductive propositions in the mechanization of λ_{RF}. (Parker et al. [36] developed a LIQUIDHASKELL metatheoretic proof but before data propositions and thus had to axiomatize a terminating evaluation relation; see § 1.2.) Refined data propositions are implemented as part of LIQUIDHASKELL's existing refined data types that already supported subtyping on constructor arguments using variant and contravariant rules, as described but not formalized in [23]. The essential extension to support data propositions is that by refining the output types of inductive data types, LIQUIDHASKELL can support constructive derivation-tree-style proofs. To use this feature in practice, we had to extend the refinement logic of LIQUIDHASKELL to use existing SMT support to make data constructors *injective*, *i.e.* if C is a constructor then $\forall x, y. C(x) = C(y) \Rightarrow x = y$. Thus, refined data types and injectivity are the two required components to implement data propositions.

Acknowledgements for Chapter 6

This chapter is adapted from "Mechanizing Refinement Types" in the proceedings of the 51st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2024), by Michael Borkowski, Niki Vazou, and Ranjit Jhala. The dissertation author was the primary investigator and author of this material.

Chapter 7

Implementation and Mechanization

7.1 LIQUIDHASKELL Mechanization

We mechanized type safety (Theorem 5.3) of λ_{RF} in both COQ 8.15.1 and LIQUIDHASKELL 8.10.7.1 (available online as supplementary material). In LIQUIDHASKELL we use refined data propositions (§ 6.2) to specify the static (*e.g.* typing, subtyping, well-formedness) and dynamic (*i.e.* small-step transitions and their closure) semantics of λ_{RF}. The LIQUIDHASKELL mechanization is simplified by SMT-automation (§ 8.1.1), uses a co-finite encoding for reasoning about variables (§ 8.1.2), and consists of proofs implemented as recursive functions that construct evidence to establish propositions by induction (§ 8.1.3).

Other that the development of data propositions, we extended LIQUIDHASKELL with two more features during the development of this proof. First, we implemented an interpreter that critically dropped the verification time from 10 hours to only 29 minutes (§7.1.1). Second, we implemented a (COQ-style) strictly-positive-occurrence checker to ensure data propositions are well-defined, since early versions of our proof used negative occurrences.

Note that while Haskell types are inhabited by diverging $\bot$ values, LIQUIDHASKELL's totality, termination, and type checks ensure that all cases are handled, the induction (recursion)

is well-founded, and that the proofs (programs) indeed inhabit the propositions (types).

7.1.1 Quantitative Results

We provide a mechanically checked proof of the type safety in § 5.2, that only assumes the requirements 3.2 and 3.3. Concretely, we assumed the primitives Requirement 3.2 for some constants of λ_{RF} because it was too strenuous to mechanically prove without interactive aid. In LIQUIDHASKELL type denotations (of Figure 3.8) cannot be currently encoded: since they include $\forall$-quantification they could only be encoded as data propositions, but the strictly-positive-occurrence checker rejects the definition of the function denotation. Due to this limitation, we can neither define the denotational implementation of the implication (§ 3.3.4.3) nor prove the denotational soundness (Theorem 5.1).

Representing Binders One main challenge in the mechanized metatheory is the syntactic representation of variables and binders [2]. The *named* representation has severe difficulties because of variable capturing substitutions and the *nameless* (*a.k.a.* de Bruijn) requires heavy index shifting. The variable representation of λ_{RF} is *locally nameless representation* [40, 3], where free variables are named, but bound variables are represented by de Bruijn indices. Our mechanization still resembles the paper and pencil proofs (performed before mechanization), yet it clearly addresses the following two problems with named bound variables. First, when different refinements are strengthened (as in Figure 3.4) the variable capturing problem reappears because we are substituting underneath a binder. Second, subtyping usually permits alpha-renaming of binders, which breaks a required invariant that each λ_{RF} derivation tree is a valid λ_F tree after erasure.

Representing Binders In our mechanization, we use the *locally-nameless representation* [3, 9]. Free variables and bound variables are taken to be separate syntactic objects, so we do not need to worry about alpha renaming of free variables to avoid capture in substitutions. We also

Table 7.1: Quantitative mechanization details. We split each development into sets of modules pertaining to regions of Figure 5.1 and for each we count lines of specification (definitions, lemma statements) and of proof.

	LIQUIDHASKELL Mechanization				COQ Mechanization		
Subject	**Files**	**Time (m)**	**Spec**	**Proof**	**Files**	**Spec**	**Proof**
Definitions	6	1	1805	374	7	941	190
Basic Properties	8	4	646	2117	8	1201	2360
λ_F Soundness	4	3	138	685	4	173	773
Weakening	4	1	379	467	4	110	568
Substitution	4	7	458	846	4	158	859
Exact Typing	2	4	70	230	2	33	182
Narrowing	1	1	88	166	1	54	262
Inversion	1	1	124	206	1	57	258
Primitives	3	4	120	277	3	89	508
λ_{RF} Soundness	1	1	14	181	1	12	233
Denotational Soundness	-	-	-	-	13	815	3010
Total	35	29	3842	5549	49	3643	9203

use de Bruijn indices only for bound variables. This enables us to avoid taking binder names into account in the strengthen function used to define substitution (Figure 3.4).

Table 7.1 summarizes the development of our metatheory, which was checked using LIQUIDHASKELL 8.10.7.1 and a Lenovo ThinkPad T15p laptop with an Intel Core i7-11800H processor. Our mechanized proofs are substantial. The entire LIQUIDHASKELL development comprises over 9,300 lines across about 35 files. Currently, the whole LIQUIDHASKELL proof can be checked in 29 minutes, which makes interactive development difficult, especially compared to the COQ proof (§ 7.2) that is checked in about 60 seconds. While incremental modular checking provides a modicum of interactivity, improving the ergonomics of LIQUIDHASKELL, *i.e.* verification time and actionable error messages, remains an important direction for future work.

7.2 COQ Mechanization

Our COQ mechanization proves both type safety and denotation soundness, *i.e.* all the statements of § 5.1 and § 5.2 and serves as a comparison for the metatheoretical development abilities of the two theorem provers. In COQ, Req. 3.2 is proved (using COQ's interactive development) and type denotations (of Figure 3.8) are defined as recursive functions using Equations [45], which make both the definition the denotational implementation of the implication (§ 3.3.4.3) and the proof the denotational soundness (Theorem 5.1) possible. The implication judgment is axiomatized per Requirement 3.3. To fairly compare the two developments in terms of effort and ergonomics, we did not use external COQ libraries because no such libraries exist yet for LIQUIDHASKELL. Vazou et al. [52] previously compared LIQUIDHASKELL and COQ as theorem provers, but their mechanizations were an order of magnitude smaller than ours and did not use data propositions (§ 6.2), which permit constructive LIQUIDHASKELL proofs.

The source code for our mechanizations in COQ and LIQUIDHASKELL, together with instructions on how to replicate the results, are available on Zenodo [7]. Additionally, a virtual appliance for Oracle VM VirtualBox is available on Zenodo [6] to assist with replication.

Acknowledgements for Chapter 7

This chapter is adapted from "Mechanizing Refinement Types" in the proceedings of the 51^{st} ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2024), by Michael Borkowski, Niki Vazou, and Ranjit Jhala. The dissertation author was the primary investigator and author of this material.

Chapter 8

Comparison of Proof Assistants

8.1 Proving Theorems in LIQUIDHASKELL

8.1.1 SMT Solvers, Arithmetic, and Set Theory

The most tedious part in the mechanization of metatheories is the establishment of invariants about variables, for example uniqueness and freshness. LIQUIDHASKELL offers a built-in, SMT automated support for the theory of sets, which simplifies establishing such invariants.

Set of Free Variables Our proof mechanization defines the Haskell function `fv` that returns the `Set` of free variable names that appear in its argument.

```
measure fv
fv :: Expr → S.Set VName
fv (EVar x)    = S.singleton x
fv (ELam e)    = fv e
fv (EApp e e') = S.union (fv e) (fv e')
... -- other cases
```

In the above (incomplete) definition, `S` is used to qualify the standard `Data.Set` Haskell library. LIQUIDHASKELL embeds the functions of `Data.Set` to SMT set operators (encoded as a map to booleans). For example, `S.union` is treated as the logical set union operator $\cup$. Further, we lift `fv` into the refinement logic using the `measure fv` annotation. The measure definition defines the logical function `fv` in the logic in a way that lets the SMT solver reason about the semantics of `fv` in a *decidable* fashion, as an uninterpreted function refining the type of each `Expr` data constructor [23]. This embedding, combined with the SMT solver's support for the theory of sets, lets LIQUIDHASKELL prove properties about expressions' free variables "for free".

Intrinsic Verification For an example of properties for free, consider the function `subFV x vx e` which substitutes the variable `x` with `vx` in `e`. The refinement type of `subFV` describes the free variables of the result.

```
subFV :: x:VName → vx:{Expr | isVal vx } → e:Expr
    → {e':Expr | fv e' ⊆ (fv vx ∪ (fv e \ x)) && (isVal e ⇒ isVal e')}
subFV x vx (EVar y)    = if x == y then vx else EVar y
subFV x vx (ELam   e)  = ELam (subFV x vx e)
subFV x vx (EApp e e') = EApp (subFV x vx e) (subFV x vx e')
... -- other cases
```

The refinement type specifies that the free variables after substitution is a subset of the free variables in the two argument expressions, excluding `x`, *i.e.* $\texttt{fv}(e[v_x/x]) \subseteq \texttt{fv}(v_x) \cup (\texttt{fv}(e) \setminus \{x\})$. This specification is proved *intrinsically*, *i.e.* the definition of `subFV` is the proof (no user aid is required) and, importantly, the specification is automatically established each time the function `subFV` is called without any need for explicit hints. The specification of `subFV` above shows another example of SMT-based proof simplification. It intrinsically proves that the value property is preserved by substitution, using the Haskell boolean function `isVal` that defines when an expression is a *value*.

Freshness LIQUIDHASKELL's support for sets simplifies defining a `fresh` function, which is often challenging[1]. `fresh xs` returns a variable that provably does not belong to its input `xs`.

```
fresh :: xs:S.Set VName → { x:VName | x ∉ xs }
fresh xs = n ? above_max n xs'
  where n    = 1 + maxs xs'
        xs'  = S.fromList xs

maxs :: [VName] → VName
maxs []     = 0
maxs (x:xs) = if maxs xs < x then x else maxs xs

above_max :: x:VName → xs:{[VName]|maxs xs < x} → {x ∉ elems xs}
above_max _ []     = ()
above_max x (_:xs) = above_max x xs
```

The `fresh` function returns `n`: the maximum element of the set increased by one. To compute the maximum element we convert the set to a list and use the inductively defined `maxs` functions. To prove `fresh`'s intrinsic specification we use an extrinsic, *i.e.* explicit, lemma `above_max n xs'` that, via the `(?)` combinator of type `a → b → a`, tells LIQUIDHASKELL that `n` is not in the set `xs`. This extrinsic lemma is itself trivially proved by induction on `xs` and SMT automation.

8.1.2 Co-finite Quantification

To encode the rules that need a fresh free variable name we use the co-finite quantification of Aydemir et al. [3], as discussed in § 3.3. Figure 8.1 presents this encoding using the T-ABS

[1]For example, COQ cannot fold over a set, and so a more complex combination of tactics is required to generate a fresh name.

```
-- Standard Existential Rule
TAbsEx :: γ:Env → tx:Type → e:Expr → t:Type
       → y:{VName | y ∉ dom γ }
       → HasTy ((y,tx):γ) (unbind y e) (unbindT y t)
       → HasTy γ (ELam e) (TFunc tx t)

-- Co-finitely Quantified Rule
TAbs  :: γ:Env → tx:Type → e:Expr → t:Type → l:S.Set VName
      → (y:{VName|y ∉ l} → HasTy ((y,tx):γ) (unbind y e) (unbindT y t))
      → HasTy γ (ELam e) (TFunc tx t)

-- Note: All rules also include kx:Kind → WfType γ tx kx elided for
   clarity.
```

Figure 8.1: Encoding of Co-finitely Quantified Rules.

rule as an example. The standard abstraction rule (rule T-ABS-EX in § 3.3) requires to provide a concrete fresh name, which is encoded in the second line of `TAbsEx` as the `y:{VName | y ∉ dom γ}` argument. The co-finitely quantified encoding of the rule `TAbs`, instead, states that there exists a specified finite set of excluded names, namely `l`, and requires that the sub-derivation holds for any name `y` that does not belong in `l`. That is, the premise is turned into a function that, given the name `y`, returns the sub-derivation. This encoding greatly simplifies our mechanization, since the premises are no more tied to concrete names, eliminating the need for renaming lemmas. We will often take `l` to be the domain of the environment, but the ability to choose `l` gives us the flexibility when constructing derivations to exclude additional names that clash with another part of a proof.

8.1.3 Inductive Proofs as Recursive Functions

The majority of our proofs are by induction on derivations. These proofs are recursive Haskell functions that operate over refined data propositions. LIQUIDHASKELL ensures the proofs are valid by checking that they are inductive (*i.e.* the recursion is well-founded), handle all

cases (*i.e.* the function is total), and establish the desired properties (*i.e.* witnesses the appropriate proposition).

Preservation (Lemma 5.5) relates the `HasTy` data proposition of § 6.2 with a `Step` data proposition that encodes Figure 3.3 and is proved by induction on the type derivation tree. Below we present a snippet of the proof, where the subtyping case is by induction while the primitive case is impossible:

```
preservation :: e:Expr → t:Type → e':Expr → HasTy Empty e t
             → Step e e' → HasTy Empty e' t
preservation _e _t e' (TSub Empty e t' t e_has_t' t'_sub_t) e_step_e'
  = TSub Empty e' t' t (preservation e t' e' e_has_t' e_step_e') t'
 _sub_t
preservation  e _t e' (TPrim _ _) step
  = impossible "value" ? lemValStep e e' step -- e ↪ e' ⇒ ¬(isVal e)
...
impossible :: {v:String | false} → a
lemValStep :: e:Expr → e':Expr → Step e e' → {¬(isVal e)}
```

In the `TSub` case we note that LIQUIDHASKELL knows that the argument `_e` is equal to the subtyping parameter `e`. The termination checker ensures the inductive call happens on a smaller derivation subtree. The `TPrim` case is by contradiction since primitives cannot step: we proved values cannot step in the `lemValStep` lemma, which is combined via the `(?)` combinator of type `a → b → a` with the fact that `e` is a value to allow the call of the false-precondition `impossible`.

LIQUIDHASKELL's totality checker ensures all cases of `HasTyEv` are covered and the termination checker ensures the proof is well-founded.

Progress (Theorem 5.4) ensures that a well-typed expression is a value *or* there exists an expression to which it steps. To express this claim we used Haskell's `Either` to encode

disjunction that contain pairs (refined to be dependent) to encode existentials.

```
progress :: e:Expr →  t:Type →  HasTy Empty e t
                   →  Either {isVal e}  (e'::Expr, Step e e')
progress _ _ (TSub Empty e t' t e_has_t' _) = progress e t' e_has_t'
progress _ _ (TPrim _ _)                    = Left ()
progress _ _ (TAbs {})                      = Left ()
...
```

The proofs of the `TSub` and `TPrim` cases are easily done by, respectively, an inductive call and establishing is-Value. The more interesting cases require us to case-split on the inductive call in order to get access to the existential witness.

8.2 COQ *vs.* LIQUIDHASKELL

COQ has a tiny trusted code base (TCB) and strong foundational mechanized soundness guarantees [46]. In contrast, LIQUIDHASKELL trusts the Haskell compiler (GHC), the SMT solver (Z3), and its constraint generation rules which have not been formalized. This work, λ_{RF}, serves precisely that purpose: by formalizing and mechanizing a significant subset of LIQUIDHASKELL, leaving out literals, casts, and data types. As far as the user experience is concerned, COQ metatheoretical developments are much faster to check, which was expected since LIQUIDHASKELL comes with expensive inference, and can be aided by relevant libraries. The two tools come with different kinds of automation: tactics *vs.* SMT, which we found to be useful in *complementary* parts of the proofs, pointing the way to possible improvements for both verification styles. Finally, LIQUIDHASKELL facilitates reasoning over mutually defined and partial functions. We begin by looking at aspects of mechanized metatheory in COQ that are easier or more feasible than in LIQUIDHASKELL, and then we turn to aspects that are easier in

LIQUIDHASKELL.

Negative Occurrences and COQ's Equations Our original LIQUIDHASKELL mechanization defined denotations as refined data propositions and proved denotational soundness. Though, we realized that the definition of the function type denotation has a negative occurrence and permitting negative occurrences can, in general, lead to unsoundness [11]. Our mechanization is the first big-scale user of LIQUIDHASKELL's data propositions thus it was not surprising that it revealed this potential unsoundness. To remove this source of unsoundness in LIQUIDHASKELL, we implemented a COQ-style positivity checker that unsurprisingly rejected the type denotation definitions. A similar challenge appears in the proof of strong normalization of the simply-type lambda calculus that because of negative occurrences cannot use inductive propositions [39]. There, the solution is to use a recursive function `expr` → `type` → `Prop` because a definition doesn't need to be computable. In our COQ mechanization, we followed a similar solution, but since our definition was not structurally recursive and was needed for the proofs, we used the full power of COQ's Equations [45] to define the type denotations. Unfortunately, a similar approach cannot currently carry over to LIQUIDHASKELL because all Haskell functions must be computable and all LIQUIDHASKELL annotations must be decidable. Therefore, quantifiers are neither allowed on the right-hand side of Haskell definitions nor in the refinements.

Tactics and Automation COQ's tactics and automation often permit shorter proofs as lemmas and constructors can be used with the `apply` tactic without writing out all arguments. For example, in LIQUIDHASKELL safety (Theorem 5.3) is encoded using Haskell's `Either` for disjunction and dependent pairs for existentials. (`Steps` is defined, using data propositions, as the closure of `Step`.)

```
safety :: e0:Expr → t:Type → e:Expr → HasTy Empty e0 t
       → Steps e0 e → Either {isVal e} (ei::Expr, Step e ei)
safety _e0 t _e e0_has_t e0_evals_e = case e0_evals_e of
  Refl e0 → progress e0 t e0_has_t        -- e0 = e
```

```
    AddStep e0 e1 e0_step_e1 e e1_eval_e →    -- e0 ↪ e1 ↪* e
      safety e1 t e (preservation e0 t e0_has_t e1 e0_step_e1) e1_eval_e
```

The reflexive case is proved by `progress`. In the inductive case the evaluation sequence is $e_0 \hookrightarrow e_1 \hookrightarrow^* e$ and the proof goes by induction, using preservation to ensure that e_1 is typed. In COQ safety is proved without any of the three fully applied calls above:

```
Theorem safety : forall (e0 e:expr) (t:type),
   Steps e0 e →  HasTy Empty e0 t →  isVal e \/ exists ei, Steps e ei.
Proof. intros; induction H.
  - (* Refl *) apply progress with t; assumption.
  - (* Add  *) apply IHSteps; apply preservation with e; assumption. Qed.
```

Automation tactics could make this proof even shorter, but we retain the essential proof structure.

Mutual Recursion LIQUIDHASKELL makes it easy to define and work with mutually recursive data types, such as our typing and subtyping judgments, and to prove mutually inductive lemmas. Similarly, our expressions, types, and predicates are three mutually recursive data types. Mutually recursive types are not a natural fit for COQ: the automatically generated induction principles do not work, so we need to use the `Scheme` keyword to generate suitable principles. Theorems involving these types cannot be broken up into separate lemmas for each type involved. Rather, one combined statement must be given, which is difficult to use in the `rewrite` tactic.

Another weakness of COQ is that all information about the hypothesis is lost during the induction tactic, so the normal structural `induction` tactic only works when a judgment contains no information, *i.e.* the data constructor is instantiated solely with universally quantified variables. For instance, in the proof of the weakening Lemma 5.9, to do structural induction on `HasTy (concat g g')e t` we must introduce a universally quantified variable `g0` and strengthen the theorem with the hypothesis `g0 = concat g g'`. While the standard library contains an "experimental" tactic `dependent induction`, we also need to work with the special mutual

induction principles that we generate for our types, so we have to directly instantiate the principle with a strengthened, complex hypothesisand state the lemma as:

```
Lemma lem_weaken_typ' : ( forall (g0 : env) (e : expr) (t : type),
  HasTy g0 e t → ( forall (g g' : env) (x : vname) (t_x : type),
     g0 = concatE g g' → unique g → unique g' →
     (binds g) ∩ (binds g') = empty → ~ (in_env x g) → ~ (in_env x g')
     → HasTy (concatE (Cons x t_x g) g') e t ) ) /\ (
forall (g0 : env) (t : type) (t' : type),
  Subtype g0 t t' → ( forall (g g' : env) (x : vname) (t_x : type),
     g0 = concatE g g' → unique g → unique g' →
     (binds g) ∩ (binds g') = empty → ~ (in_env x g) → ~ (in_env x g')
     → Subtype (concatE (Cons x t_x g) g') t t') ).
```

By contrast, in LIQUIDHASKELL we can state two separate mutually recursive lemma functions for weakening: one for typing and one for subtyping. Then we may call either lemma in their own proofs on any smaller instance of the typing (resp. subtyping) judgment. In practice, developments in COQ sidestep some of these issues by collapsing the language of terms, types, *etc.* into a single inductive data type. This approach has the advantage of reducing the number of substitution operations, but allows highly ungrammatical combinations like `App Bool False` into our syntax. We could still use this approach combined with a pre-term encoding common in COQ developments, but we preferred to keep a closer comparison to the LIQUIDHASKELL mechanization.

Partial Functions LIQUIDHASKELL facilitates the definition of partial Haskell functions and proves totality with respect to the refined types, usually automatically, without having to reason about impossible cases in mechanized proofs. For instance, our syntax does not contain an explicit `error` value, so we only want the function $\delta(c, v)$ to be defined where $c\ v$ can step in our semantics. This is straightforward in LIQUIDHASKELL: we define a predicate `isCompat ::`

`Prim → Value → Bool` and refine the input types of δ to satisfy `isCompat`. In Coq a more roundabout approach is needed: we have to define `isCompat` as an inductive type and include this object as an explicit argument to our δ function:

```
Inductive isCompat : prim →  expr →  Set : =
  | isCpt_And  : forall b, isCompat And (Bc b)
  | isCpt_Or   : forall b, isCompat Or  (Bc b)
  ...
```

However, this makes it harder to prove the determinism of our semantics due to the dependence on the proof object. One solution would be to define a partial version of δ with type `Prim → Expr → option Expr` and prove the two functions always agree regardless of proof object, *e.g.* using *subset types*; but since each value comes wrapped with a term-level proof object, agreement proofs would require a *Proof Irrelevance* axiom.

Acknowledgements for Chapter 8

This chapter is adapted from "Mechanizing Refinement Types" in the proceedings of the 51st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2024), by Michael Borkowski, Niki Vazou, and Ranjit Jhala. The dissertation author was the primary investigator and author of this material.

Chapter 9

Lists: The Language λ_{RFD}

One of the key features of GHC's core calculus missing from λ_{RF} is data types. The addition of data types would also enable us to replace the ad hoc kind system of λ_{RF} with a more versatile system of type classes. As a first step in this direction, we augment our calculus with polymorphic refined list types. We also add a measure `length` that describes the length of our lists.

9.1 Syntax and Semantics

We present the syntax and semantics of λ_{RFD} in terms of the additions to λ_{RF}. The reader may refer to the figures in Chapter 3 for the syntactic forms and rules that are inherited from λ_{RF}. As before, we use the gray to highlight the extensions to λ_F needed to support refinements in λ_{RFD}.

Constants, Values, and Terms Figure 9.1 summarizes the syntax of terms in both λ_{RFD} and in the unrefined calculus. The *primitives c* now include a `succ` function on integers that adds one and a (polymorphic) `len` measure that computes the length of any list. Although a user could easily define `len` using `switch` below, we add `len` as a built-in primitive in order to use it in our typing judgments. Our *terms e* now additionally contain two list constructors: the empty list

Primitives	c	$::=$	$\cdots$	
		$\mid$	`succ`	*integer ops.*
		$\mid$	`len`	*polymorphic list ops*
List Values	ℓ	$::=$	`nil` $[t]$	*empty list*
		$\mid$	`cons` $[t]\ v\ \ell$	*list constructor*
Values	v	$::=$	$\cdots$	
		$\mid$	ℓ	*list values*
Terms	e	$::=$	$\cdots$	
		$\mid$	`cons` $[t]\ e_1\ e_2$	*list constructor*
		$\mid$	`switch` $(e)\ e_n\ e_c$	*list destructor*

Figure 9.1: Syntax of Primitives, Values, and Expressions.

Types	t	$::=$	$\cdots$	
		$\mid$	$[t]\{\nu : p\}$	*refined list type*

Figure 9.2: Syntax of Types. The gray boxes are the extensions to λ_F needed by λ_{RFD}.

`nil` $[t]$ and the non-empty `cons` $[t]\ e_1\ e_2$, which builds a list from a head element and a tail. Both of these constructors take a refined type annotation. For instance, we should be able to type check

$$\texttt{cons}\ [\texttt{Int}\{\nu : \nu \geq 0\}]\ 1\ (\texttt{cons}\ [\texttt{Int}\{\nu : \nu > 0\}]\ 2\ 3)$$

but not

$$\texttt{cons}\ [\texttt{Int}\{\nu : \nu > 0\}]\ 1\ (\texttt{cons}\ [\texttt{Int}\{\nu : \nu \geq 0\}]\ 2\ 3)$$

because the tail of the latter list is only known to consist of non-negative integers. Although we mechanized the metatheory in the same manner without type annotations as well, these annotations are needed to enable future work on a bidirectional type checking algorithm (Chapter 10). The terms also now contain a list destructor `switch` $(e)\ e_n\ e_c$, which case splits on the shape of the match scrutinee e. Finally, *values* v are augmented by lists that contain only values as elements; these list values are defined inductively in Figure 9.1.

Kinds & Types Figure 9.2 shows the syntax of the types, with the gray boxes indicating the extensions to λ_F required by λ_{RFD}. In contrast λ_{RF}, our list types are not base types, but they

Operational Semantics (ext. Figure 3.3) $\boxed{e \hookrightarrow e'}$

$$\frac{e \hookrightarrow e'}{\texttt{cons}\ [t]\ e\ e_1 \hookrightarrow \texttt{cons}\ [t]\ e'\ e_1}\text{E-PLCons} \qquad \frac{e \hookrightarrow e'}{\texttt{cons}\ [t]\ v\ e \hookrightarrow \texttt{cons}\ [t]\ v\ e'}\text{E-PRCons}$$

$$\frac{e \hookrightarrow e'}{\texttt{switch}\ (e)\ e_n\ e_c \hookrightarrow \texttt{switch}\ (e')\ e_n\ e_c}\text{E-PSwitch} \qquad \frac{}{\texttt{switch}\ (\texttt{nil}\ [t])\ e_n\ e_c \hookrightarrow e_n}\text{E-SwitchN}$$

$$\frac{}{\texttt{switch}\ (\texttt{cons}\ [t]\ v_1\ v_2)\ e_n\ e_c \hookrightarrow (e_c\ v_1)\ v_2}\text{E-SwitchC}$$

Figure 9.3: The small-step semantics for λ_{RFD}.

can be refined. Both of these are key to data types: our lists may contain incomparable data such as lambda abstractions, and so they cannot support the polymorphic comparison operators within our simple kind system. However, refinements on lists are key to any model of data types. We must be able to express the type of a program such as the following safe tail function:

```
tail :: forall a. {v:[a] | length v > 0 } → [a]
tail xs = switch (xs) error (\y ys → ys)
```

λ_{RFD} keeps the simple kind system from λ_{RF} and enforces list types as $\star$-kinded to prevent the substitution of a list type for a refined type variable. This prevents a refinement of a list type from attempting to compare a list using one of the polymorphic list operators.

Dynamic Semantics Figure 9.3 summarizes the small-step semantics for both calculi.

Typing and Well-formedness Next, we present the static semantics of λ_{RFD} by describing the additional rules used to establish our well-formedness, typing, and subtyping judgments. Figure 9.4 summarizes the new rules that establish the well-formedness of types. Rule WFLIST states that a list type $[t]\{x:\texttt{true}\}$ with empty refinement is well-formed with star kind provided that t is well-formed with some kind k. Similar to rule WF-REFN, our rule WF-LISTR stipulates that a refined list type $[t]\{x:p\}$ is well-formed with star kind in some environment if the trivially refined type $[t]\{x:\texttt{true}\}$ has star kind in the same environment and if the refinement predicate p

Well-formed Type (ext. Figure 3.5) $\boxed{\Gamma \vdash_w t : k}$

$$\frac{\Gamma \vdash_w t : k}{\Gamma \vdash_w [t]\,\{x : \texttt{true}\} : \star}\text{WF-LIST} \qquad \frac{\Gamma \vdash_w [t]\{x : \texttt{true}\} : \star \quad \forall y \notin \Gamma.y : [t], \lfloor \Gamma \rfloor \vdash_F p[y/x] : \texttt{Bool}}{\Gamma \vdash_w [t]\{x : p\} : \star}\text{WF-LISTR}$$

Figure 9.4: Well-formedness of λ_{RFD} types. The rules for λ_F exclude the gray boxes.

has type `Bool` in the environment augmented by binding a fresh variable to type $[t]$.

Figure 9.5 summarizes the rules that establish typing for both λ_F and λ_{RF}, with gray for the λ_{RF} extensions. Rule T-NIL states that whenever t is a well-formed type in some environment, then `nil` $[t]$ has the type $[t]\{x : \texttt{len}\, x = 0\}$ of lists of elements of type t of length zero. The rule T-CONS is slightly more complex: whenever e_h can be given type t in some environment, and whenever e_t can be given type $[t]\{x : p\}$ in the same environment, then the list `cons` $[t]$ e_h e_t can be given the type $[t]$ with the refinement that says that this list has length one more than some list of type $[t]\{x : p\}$. The purpose of this refinement is to embed the information about the specific length of a list at the refinement level. Our typing rule for the list destructor `switch` (e) e_n e_c is best thought of as analogous to our rule T-IF because it enables path-sensitive reasoning about lists. This rule T-SWITCH says that whenever the match scrutinee e can be given a list type $[t]\{x : p\}$ in some environment, whenever t' is well-typed in the same environment, and whenever each branch can be given this type t' in this environment augmented by the knowledge that the about the scrutinee and its length, then the full term `switch` (e) e_n e_c can be given type t'. Note that, per the semantics in 9.3, the cons-branch e_c is a function expecting two arguments: the head of the scrutinee and the tail (which we know has length one less than the scrutinee). In our formalism, it is necessary for us to augment the environment with *two* dummy variables here. We need to preserve p, the knowledge obtained from the match scrutinee, and the fact that the scrutinee is one element longer than the tail to which e_c is being applied. We could express the

Typing (ext. Figure 3.6) $\boxed{\Gamma \vdash e : t}$

$$\frac{\Gamma \vdash_w t : k}{\Gamma \vdash \texttt{nil}\ [t] : [t]\ \{x : \texttt{len}\ x = 0\}}\text{T-NIL}$$

$$\frac{\Gamma \vdash e_h : t \qquad \Gamma \vdash e_t : [t]\ \{x : p\}}{\Gamma \vdash \texttt{cons}\ [t]\ e_h\ e_t : \exists y{:}[t]\ \{x : p\}\ .\ [t]\ \{\nu : \texttt{len}\ \nu = \texttt{succ}\ \texttt{len}\ y\}}\text{T-CONS}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : [t]\ \{x : p\} \qquad \Gamma \vdash_w t' : k \\ \forall y \notin \Gamma.\ \ y : [t]\{x : p \wedge \texttt{len}\ x = 0\}, \Gamma \vdash e_n : t' \\ \forall y, z \notin \Gamma.\ \ z : [t]\{x : p \wedge \texttt{succ}\ \texttt{len}\ y = \texttt{len}\ x\}, y : [t], \Gamma \vdash e_c : t \to [t]\ \{\nu : \texttt{len}\ y = \texttt{len}\ \nu\} \to t'\end{array}}{\Gamma \vdash \texttt{switch}\ (e)\ e_n\ e_c : t'}\text{T-SWITCH}$$

Figure 9.5: Typing rules. The judgment $\Gamma \vdash_F e : \tau$ is extended by excluding the gray boxes.

antecedent judgment as

$$\forall z \notin \Gamma. z : [t]\{x : p\}, \Gamma \vdash e_c : t \to [t]\ \{\nu : \texttt{succ}\ \texttt{len}\ \nu = \texttt{len}\ z\} \to t', \tag{9.1}$$

but this would pose a problem for proving type soundness for λ_{RFD} (specifically preservation) from a minimal interface of axioms (9.1) for implication. These axioms are purely syntactic, and so are not sufficient to prove implications (and hence subtyping obligations) where there are slight variations in refinement syntax. Rule T-CONS gives us a refinement that says the length of this list is one *longer* than some other list; judgment 9.1 would require us to show that the second argument to e_c has length one *shorter* than some other list. There is no way to derive this knowledge without either a semantic notion of entailment, or a large, unwieldy set of axioms, or by changing the form of rule T-SWITCH as we did.

New Primitives The function $\text{ty}(c)$, which gives the type of every built-in primitives, is extended for the new primitives `succ` and `len` . Below we present essential examples of the $\text{ty}(c)$

Subtyping $\boxed{\Gamma \vdash s \preceq t}$

$$\frac{\Gamma \vdash t_1 \preceq t_2 \quad \forall y \notin \Gamma.\ y{:}[t_1]\{\texttt{true}\}, \Gamma \vdash p_1[y/x] \Rightarrow p_2[y/x]}{\Gamma \vdash [t_1]\{x{:}p_1\} \preceq [t_2]\{x{:}p_2\}}\ \text{S-LIST}$$

Figure 9.6: Subtyping Rules.

definition.

$$\begin{aligned} \texttt{ty}(\texttt{succ}) &\doteq y{:}\texttt{Int} \to \texttt{Int}\{v{:}v = (\texttt{succ}\ y)\} \\ \texttt{ty}(\texttt{len}) &\doteq \forall \alpha{:}\star.y{:}\alpha \to \texttt{Int}\{v{:}v = (\texttt{len}\ y)\} \end{aligned}$$

For ease of reading, the `len` used in the refinements is the polymorphic `len` , but with the type applications elided.

9.1.1 Subtyping

Figure 3.7 presents the new rule to establish the subtyping judgment $\Gamma \vdash s \preceq t$. Rule S-LIST states that one list type $[t_1]\{\nu : p_1\}$ is a subtype of another list type $[t_2]\{\nu : p_2\}$ in some environment Γ, when t_1 is a subtype of t_2 and p_1 implies p_2 in the environment Γ augmented by binding a fresh type variable to kind k.

Implication In § 3.3.4 we discussed our approach to formalizing implication by giving both an axiomatized interface and a denotational implementation. First, we give the additional axioms for implication in λ_{RFD}.

Requirement 9.1 (Implication Interface)**.** *The implication relation satisfies the statements in Requirement 3.3 and the statements below:*

1. *(Exact Quantification) If* $\Gamma \vdash v_x : t_x$ *and* $\Gamma \vdash_w t_x : B$ *and* $x{:}\mathsf{self}(t_x, v_x, B) \in \Gamma$ *and* $x \notin \mathsf{free}(v_x)$ *then* $\Gamma \vdash p \Rightarrow p[v_x/x]$.

2. *(Equal Length Quantification) If* $\Gamma \vdash v : [t]\{x{:}p\}$ *and* $\Gamma \vdash_w [t]\{x{:}p\} : \star$ *and*

$x : [t]\{x : \texttt{len}\ x = \texttt{len}\ v \wedge p\} \in \Gamma$ *and* $x \notin \mathsf{free}(v)$ *and* $\mathsf{safeListVar}(x, q)$ *then* $\Gamma \vdash q \Rightarrow q[v/x]$ *and* $\Gamma \vdash q[v/x] \Rightarrow q$.

The first statement says that whenever $x : \mathsf{self}(t_x, v_x, B)$ appears bound in Γ, then x is effectively being universally quantified over a type with just one inhabitant (up to equality). Then we assume that a refinement p implies p with all occurrences of x replaced by v_x. This is semantically valid because any encoding of our refinements in an external logic (such as SMT) would map $=$ in our refinement syntax to equality. And we also prove that under the denotational definition of implication (§ 3.3.4.3) our syntactic primitive $=$ actually corresponds to semantic equality, and we are thus able to show that this axiom follows from the denotational definition.

The second statement, which we call Equal Length Quantification, expresses the best possible analogue for lists. We cannot appeal to Exact Quantification for any variable x bound to a list type in the environment; these types cannot be selfified because lists cannot be compared, even for equality, in our system. However, the refinements that do occur in our system (by virtue of appearing in our various rules) do not use the `switch` statement to destruct lists to inspect their contents. Rather, these refinements are entirely agnostic about the data contained in lists and are only concerned with the length of lists. We capture this notion in the recursive function $\mathsf{safeListVar}(x, q)$ which states that x only appears in q as the argument to the `len` function. If this is the case, then the only information that q uses about x is the length. The Exact Length Quantification axiom then says that if x is bound to a type that constrains the length of x to be equal to the length of a list v, then the refinement q is equivalent (under implication) to $q[v/x]$.

Previously, we noted in § 3.3.4.1 that we did not require the Exact Quantification axiom in Assumption 1 of [26] to formalize implication in λ_{RF}. However, we do require it for λ_{RFD} in our mechanization to verify Requirement 3.2 as it applies to the length of lists.

$$[\![\,[t]\{x:p\}\,]\!] \doteq \{v \mid \varnothing \vdash_F v : [\lfloor t \rfloor] \wedge (\forall v_i \in v.\, v_i \in [\![\,t\,]\!]) \wedge p[v/x] \hookrightarrow^* \texttt{true}\}.$$

Figure 9.7: Denotations of Types and Environments.

9.1.2 Denotational Semantics

We extend the definition of denotations of types given in Figure 3.8 to define the denotation of a list type. In order to define the denotation $[\![\,[t]\{x:p\}\,]\!]$ as the set of closed values of the appropriate base type $[\lfloor t \rfloor]$ which satisfy the type's refinement predicate, we have to take into account that there are at least two refinements here. The entire list v must satisfy the refinement p, and additionally the type t itself may contain refinements which must be satisfied by each of the elements of v.

9.2 Metatheory of Lists

Our proof of type soundness of λ_{RFD} generally follows the same structure as the soundness proof for λ_{RF}, illustrated in Figure 5.1. However, some lemmas have additional cases that are highly non-trivial and add additional difficulty to the proof. For instance, we need to be able to invert typing judgments such as $\Gamma \vdash \texttt{cons}\ [t]\ e_h\ e_t : t'$ both to obtain typing judgments for e_h and e_t and also to obtain a typing judgment that contains the knowledge about the length of $\texttt{cons}\ [t]\ e_h\ e_t$, which may have been discarded through subsumption.

We give the highlights for the metatheory of λ_{RFD}.

Inversion of Typing Judgments In § 5.3 we discussed Inversion Lemma 5.7, which allowed us to invert the typing judgment for a term- or type-abstraction. We need to extend this lemma to include inverting typing judgments for lists. We can then recover typing information about the head and tail of this list as well as information about the length of the list that may have been lost through use of the subsumption T-SUB rule.

Lemma 9.2. *(Inversion of* T-NIL, T-CONS*) (extends Lemma 5.7)*

1. *If* $\Gamma \vdash \texttt{nil}\ [t_0] : [t]\{x : p\}$ *and* $\vdash_w \Gamma$*, then* $\Gamma \vdash t_0 \preceq t$*, and* $\Gamma \vdash_w t_0 : \star$*, and* $\Gamma \vdash [t_0]\{x : \texttt{len}\, x = 0\} \preceq [t]\{x : p\}$.

2. *If* $\Gamma \vdash \texttt{cons}\ [t_0]\ v_1\ v_2 : [t]\{x : p\}$ *and* $\vdash_w \Gamma$*, then* $\Gamma \vdash t_0 \preceq t$*,* $\Gamma \vdash_w t_0 : \star$*,* $\Gamma \vdash v_1 : t_0$ *and for some refinement* q*,* $\Gamma \vdash v_2 : [t_0]\{x : q\}$ *and* $\Gamma \vdash \exists y{:}[t]\{x : q\}.\, [t_0]\{x : \texttt{len}\, x = \texttt{succ}\ \texttt{len}\, y\} \preceq [t]\{x : p\}$.

The second statement above is important because if $\Gamma \vdash \texttt{cons}\ [t_0]\ v_1\ v_2 : [t]\{x : p\}$ then we cannot get directly to a typing judgment for v_1 or for v_2. Indeed, rule T-CONS gives us an existentially quantified list of type $[t_0]$ with a refinement relating the length of the list to its tail; but here t may not even be the same as t_0, so the derivation tree must have used one or more applications of rule T-SUB. The proof goes by induction on the size of the derivation tree, which must be finite.

Exact Typing Although we cannot apply the exact typing lemma to lists (only base types can be selfified), we can derive an analogous statement in terms of equality of length. whenever we can type a list value v at type $[t]\{x : p\}$ then we also type v at the type $[t]\{x : p\}$ with the refinement strengthened by $\texttt{len}\, x = \texttt{len}\, v$.

Lemma 9.3. *(Equal Length in Typing, compare to Lemma 5.11) If* $\Gamma \vdash v : [t]\{x : p\}$ *and* $\vdash_w \Gamma$ *then* $\Gamma \vdash v : [t]\{x : \texttt{len}\, x = \texttt{len}\, v \wedge p\}$.

9.3 Implementation

We implemented the metatheory of λ_{RFD} in COQ. This mechanization proves both type safety and denotation soundness. Compared to mechanization of λ_{RF}, this proof was about 35% longer in lines of code and takes about twice as long (two minutes) to check. This mechanization is also included as supplementary material. Table 9.1 summarizes the development of our

Table 9.1: Comparative mechanization details for λ_{RF} versus λ_{RFD}.

COQ Mechanizations

Subject	**Files**	λ_{RF} **lines**	λ_{RFD} **Lines**	**% Increase**	λ_{RFD} **Spec**	λ_{RFD} **Proof**
Definitions	7	1155	1476	27.8%	1193	283
Basic Properties	8	3626	3980	9.8%	1236	2744
λ_F Soundness	4	983	1156	17.5%	178	978
Weakening	4	719	868	20.7%	110	758
Substitution	4	1079	1248	15.7%	165	1083
Exact Typing	2	246	318	29.3%	73	245
Narrowing	1	333	381	14.4%	54	327
Inversion	1	339	813	139.8%	145	668
Primitives	2	635	1192	87.7%	137	1055
λ_{RF} Soundness	1	263	657	149.8%	12	645
Denot. Soundness	13	4033	6061	50.3%	1147	4914
Total	50	13411	18150	35.3%	4450	13700

metatheory, and points to which areas of the proof increased the most in length in λ_{RFD} as compared with λ_{RF}. Finally, in the last two columns, the table provides a breakdown for each broad area of the soundness proof of λ_{RFD} into lines of specification (definitions and theorem statements in Gallina) and lines of proof (proofs written in Ltac, COQ's tactic language).

The source code for our mechanization of in COQ of the soundness proof for λ_{RFD} is available on Zenodo [5].

Acknowledgements for Chapter 9

This chapter consists of unpublished work done in collaboration with Ranjit Jhala. This material is being prepared for future submission for publication. The dissertation author was the primary investigator and author of this material.

Chapter 10

Conclusions & Future Work

We presented and formalized, for the first time, the soundness of λ_{RF} and λ_{RFD}: the former is a refinement calculus with semantic subtyping, existential types, and parametric polymorphism, which are critical for practical refinement typing. The latter adds basic concrete data types and measures: lists and a built-in length function. Our metatheory is mechanized in both COQ and (for the core λ_{RF}) LIQUIDHASKELL, the latter using the novel feature of refined data propositions to reify derivations as (refined) Haskell datatypes, using SMT to automate invariants about variables.

While our proof can be mechanized in other proof assistants like AGDA [34], ISABELLE [33], BELUGA [37], DAFNY [30], or F* [31], our goal here is not to compare LIQUIDHASKELL against every system. Instead, our primary contribution is to, for the first time, *establish the soundness* of the combination of features critical for practical refinement typing and show that such a proof can be *mechanized as a plain program* with refinement types. ***Metatheory*** Looking ahead, we envision two lines of work on mechanizing metatheory *of* and *with* refinement types.

1. Mechanization of Refinements λ_{RF} covers a crucial but small fragment of the features of modern refinement type checkers. Building on this, λ_{RFD} takes the first step towards data types. The next step is to extend the language to include literals, casts, and arbitrary user-specified data types, thus covering *all* GHC's core calculus. Next, λ_{RF} can be extended to more sophisticated

features of refinement types, such as abstract and bounded refinements and refinement reflection. Similarly, our current work axiomatizes the requirements of the semantic implication checker (*i.e.* SMT solver). It would be interesting to implement a solver and verify that it satisfies that contract, or alternatively, show how proof certificates [32] could be used in place of such axioms.

2. Mechanization with Refinements While this work shows that non-trivial meta-theoretic proofs are *possible* with SMT-based refinement types, our experience is that much remains to make such developments *pleasant*. For example, programming would be far more convenient with support for automatically *splitting cases* or filling in *holes* as done in Agda [34] and envisioned by Redmond et al. [41]. Similarly, when a proof fails, the user has little choice but to think really hard about the internal proof state and what extra lemmas are needed to prove their goal. Finally, the stately pace of verification — 9400 lines across 35 files take about 30 minutes — hinders interactive development. Thus, rapid incremental checking, lightweight synthesis, and actionable error messages would go a long way towards improving the ergonomics of verification, and hence remain important directions for future work.

Algorithmic type checking The key motivation behind the use of liquid types in practical systems is the ability to typecheck programs decidably. When refinements are restricted to a decidable logic, then verification can be done by an SMT solver without reliance on brittle heuristics. As a next step, we aim to show that our typing system can be equivalently cast in the form of a bi-modal or "bidirectional" algorithm that combines checking typing obligations (from explicit annotations left by the programmer or at function application sites, for instance) with synthesizing types from program subterms where possible [14]. Combined with restrictions on the syntax of refinements, this would make typechecking for a significant subset of λ_{RFD} programs decidable.

Acknowledgements for Chapter 10

This chapter is adapted from "Mechanizing Refinement Types" in the proceedings of the 51st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2024), by Michael Borkowski, Niki Vazou, and Ranjit Jhala. The dissertation author was the primary investigator and author of this material.

Bibliography

[1] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers. The prusti project: Formal verification for rust (invited). In *NASA Formal Methods (14th International Symposium)*, pages 88–108. Springer, 2022. URL https://link.springer.com/chapter/10.1007/978-3-031-06773-0_5.

[2] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005. doi: 10.1007/11541868_4. URL https://doi.org/10.1007/11541868_4.

[3] Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 3–15. ACM, 2008. doi: 10.1145/1328438.1328443. URL https://doi.org/10.1145/1328438.1328443.

[4] João Filipe Belo, Michael Greenberg, Atsushi Igarashi, and Benjamin C. Pierce. Polymorphic contracts. In Gilles Barthe, editor, *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6602 of *Lecture Notes in Computer Science*, pages 18–37. Springer, 2011. doi: 10.1007/978-3-642-19718-5_2. URL https://doi.org/10.1007/978-3-642-19718-5_2.

[5] Michael H. Borkowski and Ranjit Jhala. Coq Mechanization for Chapter 9 of Dissertation "Mechanizing Refinement Types", August 2024. URL https://doi.org/10.5281/zenodo.13352164.

[6] Michael H. Borkowski, Niki Vazou, and Ranjit Jhala. Artifact Virtual Machine for "Mechanizing Refinement Types", October 2023. URL https://doi.org/10.5281/zenodo.8425176.

[7] Michael H. Borkowski, Niki Vazou, and Ranjit Jhala. Artifact for "Mechanizing Refinement Types", October 2023. URL https://doi.org/10.5281/zenodo.8425960.

[8] Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, page 198–208, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930906. doi: 10.1145/1069774.1069793. URL https://doi.org/10.1145/1069774.1069793.

[9] Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3):363–408, 2012. doi: 10.1007/s10817-011-9225-2. URL https://doi.org/10.1007/s10817-011-9225-2.

[10] Zilin Chen. A hoare logic style refinement types formalisation. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2022, page 1–14, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394390. doi: 10.1145/3546196.3550162. URL https://doi.org/10.1145/3546196.3550162.

[11] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, pages 50–66, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg. ISBN 978-3-540-46963-6. doi: https://doi.org/10.1007/3-540-52335-9_47.

[12] Benjamin Cosman and Ranjit Jhala. Local refinement typing. *Proc. ACM Program. Lang.*, 1(ICFP):26:1–26:27, 2017. doi: 10.1145/3110270. URL https://doi.org/10.1145/3110270.

[13] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3. doi: https://doi.org/10.1007/978-3-540-78800-3_24.

[14] Joshua Dunfield and Neel Krishnaswami. Bidirectional typing. 2020. https://arxiv.org/abs/1908.05839.

[15] Cormac Flanagan. Hybrid type checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, page 245–256, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595930272. doi: 10.1145/1111037.1111059. URL https://doi.org/10.1145/1111037.1111059.

[16] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, page 237–247, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915984. doi: 10.1145/155090.155113. URL https://doi.org/10.1145/155090.155113.

[17] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In *Proceedings of the 18th ACM Conference on Computer and*

Communications Security, CCS '11, page 341–350, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309486. doi: 10.1145/2046707.2046746. URL https://doi.org/10.1145/2046707.2046746.

[18] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 137–146. IEEE Computer Society, 2002. doi: 10.1109/LICS.2002.1029823. URL https://doi.org/10.1109/LICS.2002.1029823.

[19] Jad Elkhaleq Ghalayini and Neel Krishnaswami. Explicit refinement types. *Proc. ACM Program. Lang.*, 7(ICFP), aug 2023. doi: 10.1145/3607837. URL https://doi.org/10.1145/3607837.

[20] Andrew D. Gordon and C. Fournet. Principles and applications of refinement types. In *Logics and Languages for Reliability and Security*. IOS Press, 2010. URL https://doi.org/10.3233/978-1-60750-100-8-73.

[21] Michael Greenberg. *Manifest Contracts*. PhD thesis, University of Pennsylvania, 2013. URL https://repository.upenn.edu/edissertations/468/.

[22] Jad Hamza, Nicolas Voirol, and Viktor Kuncak. System FR: formalized foundations for the stainless verifier. *Proc. ACM Program. Lang.*, 3(OOPSLA):166:1–166:30, 2019. doi: 10.1145/3360592. URL https://doi.org/10.1145/3360592.

[23] Ranjit Jhala and Niki Vazou. Refinement types: A tutorial. *Found. Trends Program. Lang.*, 6 (3-4):159–317, 2021. doi: 10.1561/2500000032. URL https://doi.org/10.1561/2500000032.

[24] Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. Occurrence typing modulo theories. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 296–309, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342612. doi: 10.1145/2908080.2908091. URL https://doi.org/10.1145/2908080.2908091.

[25] Tristan Knoth, Di Wang, Adam Reynolds, Jan Hoffmann, and Nadia Polikarpova. Liquid resource types. *Proc. ACM Program. Lang.*, 4(ICFP):106:1–106:29, 2020. doi: 10.1145/3408988. URL https://doi.org/10.1145/3408988.

[26] Kenneth Knowles and Cormac Flanagan. Compositional reasoning and decidable checking for dependent contract types. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*, PLPV '09, page 27–38, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583303. doi: 10.1145/1481848.1481853. URL https://doi.org/10.1145/1481848.1481853.

[27] Nico Lehmann and Éric Tanter. Formalizing simple refinement types in Coq. In *2nd International Workshop on Coq for Programming Languages (CoqPL'16)*, St. Petersburg, FL, USA, January 2016.

[28] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. STORM: Refinement types for secure web applications. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 441–459. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/lehmann.

[29] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. Flux: Liquid types for rust. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023. doi: 10.1145/3591283. URL https://doi.org/10.1145/3591283.

[30] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010. URL https://doi.org/10.1007/978-3-642-17511-4_20.

[31] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *European Symposium on Programming (ESOP)*, 2019. URL https://doi.org/10.1007/978-3-030-17184-1_2.

[32] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 106–119, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897918533. doi: 10.1145/263699.263712. URL https://doi.org/10.1145/263699.263712.

[33] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science. 2002. URL https://link.springer.com/book/10.1007/3-540-45949-9.

[34] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers, 2007.

[35] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics*, pages 437–450, Boston, MA, 2004. Springer US. ISBN 978-1-4020-8141-5. doi: https://doi.org/10.1007/1-4020-8141-3_34.

[36] James Parker, Niki Vazou, and Michael Hicks. Lweb: information flow security for multi-tier web applications. *Proc. ACM Program. Lang.*, 3(POPL):75:1–75:30, 2019. doi: 10.1145/3290388. URL https://doi.org/10.1145/3290388.

[37] Brigitte Pientka. Beluga: Programming with dependent types, contextual data, and contexts. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming*, pages 1–12, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-12251-4. doi: https://doi.org/10.1007/978-3-642-12251-4_1.

[38] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. URL https://www.cis.upenn.edu/~bcpierce/tapl/.

[39] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*, volume 2 of *Software Foundations*. Electronic textbook, 2022. URL https://softwarefoundations.cis.upenn.edu/.

[40] Randy Pollack. Closure under alpha-conversion. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 313–332. Springer, 1993. doi: 10.1007/3-540-58085-9_82. URL https://doi.org/10.1007/3-540-58085-9_82.

[41] Patrick Redmond, Gan Shen, and Lindsey Kuper. Toward hole-driven development with liquid haskell. *CoRR*, abs/2110.04461, 2021. URL https://arxiv.org/abs/2110.04461.

[42] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 159–169, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595938602. doi: 10.1145/1375581.1375602. URL https://doi.org/10.1145/1375581.1375602.

[43] Didier Rémy. Type systems for programming languages. Course notes, 2021. URL https://www.doc.ic.ac.uk/~svb/TSfPL/.

[44] Taro Sekiyama, Atsushi Igarashi, and Michael Greenberg. Polymorphic manifest contracts, revised and resolved. *ACM Trans. Program. Lang. Syst.*, 39(1):3:1–3:36, 2017. doi: 10.1145/2994594. URL https://doi.org/10.1145/2994594.

[45] Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in coq. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019. doi: 10.1145/3341690. URL https://doi.org/10.1145/3341690.

[46] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq Correct. Verification of Type Checking and Erasure for Coq, in Coq. In *Principles of Programming Languages (POPL)*, 2020. URL https://doi.org/10.1145/3371076.

[47] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, page 53–66, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 159593393X. doi: 10.1145/1190315.1190324. URL https://doi.org/10.1145/1190315.1190324.

[48] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss,

Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent Types and Multi-Monadic Effects in F*. In *Principles of Programming Languages (POPL)*, 2016. URL https://doi.org/10.1145/2837614.2837655.

[49] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, page 395–406, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595936899. doi: 10.1145/1328438.1328486. URL https://doi.org/10.1145/1328438.1328486.

[50] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. Liquidhaskell: Experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, page 39–51, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330411. doi: 10.1145/2633357.2633366. URL https://doi.org/10.1145/2633357.2633366.

[51] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 269–282, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328739. doi: 10.1145/2628136.2628161. URL https://doi.org/10.1145/2628136.2628161.

[52] Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. A tale of two provers: Verifying monoidal string matching in LIQUIDHASKELL and COQ. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell 2017, page 63–74, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351829. doi: 10.1145/3122955.3122963. URL https://doi.org/10.1145/3122955.3122963.

[53] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: complete verification with SMT. *Proc. ACM Program. Lang.*, 2(POPL):53:1–53:31, 2018. doi: 10.1145/3158141. URL https://doi.org/10.1145/3158141.

[54] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, page 347–359, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913280. doi: 10.1145/99370.99404. URL https://doi.org/10.1145/99370.99404.