Mechanizing Refinement Types

MICHAEL H. BORKOWSKI, UC San Diego, USA
NIKI VAZOU, IMDEA Software Institute, Spain
RANJIT JHALA, UC San Diego, USA

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. Finally, 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.

CCS Concepts: • **Software and its engineering** → **Formal software verification**.

Additional Key Words and Phrases: refinement types, LiquidHaskell

ACM Reference Format:
Michael H. Borkowski, Niki Vazou, and Ranjit Jhala. 2024. Mechanizing Refinement Types. *Proc. ACM Program. Lang.* 8, POPL, Article 70 (January 2024), 30 pages. https://doi.org/10.1145/3632912

1 INTRODUCTION

Refinements constrain types with logical predicates to specify new concepts. For example, the refinement type $\texttt{Pos} \doteq \texttt{Int}\{v{:}0 < v\}$ describes *positive* integers and $\texttt{Nat} \doteq \texttt{Int}\{v{:}0 \leq v\}$ specifies natural numbers. Refinement types have been successfully used to specify various properties like secrecy [Fournet et al. 2011], resource usage [Knoth et al. 2020], or information flow [Lehmann et al. 2021] that can then be verified in programs developed in various programming languages like Haskell [Vazou et al. 2014b], Scala [Hamza et al. 2019], and Racket [Kent et al. 2016].

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 $g\ e : \texttt{Pos}$ if, for example, g is the

Authors' addresses: Michael H. Borkowski, UC San Diego, La Jolla, USA, mborkows@eng.ucsd.edu; Niki Vazou, IMDEA Software Institute, Madrid, Spain, niki.vazou@imdea.org; Ranjit Jhala, UC San Diego, USA, rjhala@ucsd.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/1-ART70
https://doi.org/10.1145/3632912

identity function $g : a \rightarrow a$. As a more interesting example, in § 2 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). Unsurprisingly, the addition of polymorphism poses further challenges. As Sekiyama et al. [2017] observe, a naïve definition of type instantiation can lose potentially contradicting refinements leading to unsoundness. Existing formalizations of refinement types drop semantic subtyping [Hamza et al. 2019; Sekiyama et al. 2017] or polymorphism [Flanagan 2006; Swamy et al. 2016], or have problematic metatheory [Belo et al. 2011].

In this paper we formalize λ_{RF}, a core calculus with a refinement type system that combines semantic subtyping with polymorphism, via four concrete contributions.

1. Reconciliation Our first contribution is 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 [Knowles and Flanagan 2009] which is essential to *synthesize* the types – in the sense of bidirectional typing – for applications and let-binders in a compositional manner (§ 3, 4).

2. Foundation Our second contribution is to establish the foundations of λ_{RF} by proving soundness, which says that well-typed expressions cannot get stuck and belong in the denotation of their type (§ 5). 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 [Pierce 2002], in rules where refinements are not required, cutting two potential circularities in the static judgments (Figure 2). Second, we define an *implication interface* that abstractly specifies the properties of implication required to prove type soundess, and show how this interface can be implemented via denotational semantics (§ 4.4).

3. Reification Our third contribution is to introduce *data propositions*, a novel feature 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). 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 Our final contribution is to mechanize the metatheory of λ_{RF} *twice*: using LiquidHaskell and Coq. We formalized λ_{RF} in LiquidHaskell (§ 7) 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 (§ 8) 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 [Sozeau and Mangin 2019]. Further, the Coq development is much faster (about 60 seconds to verify), but more difficult 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.

```
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
```

Fig. 1. Functional Arrays with refinement types that ensure safe indexing.

2 REFINEMENT TYPES

We start by an informal overview of the refined core calculus λ_{RF}, which we later present formally (§ 3) and prove sound (§ 5). Concretely, we present the goals of refinement types (§ 2.1) and how they are achieved via the three essential features of 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 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. 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 7) 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 1 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 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 [Rondon et al. 2008] 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 § 4 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. 1), the type of 4 was implicitly converted. To see how, consider an environment Γ that contains the array interface. Let $\Gamma \subseteq \{\texttt{get} : n : \texttt{Int} \rightarrow i : \texttt{Int}\{v : v < n\} \rightarrow \texttt{ArrayN}\ a\ n \rightarrow a\}$ For brevity, we ignore the requirement that i and n are natural numbers and, as in Fig. 1, we use $\texttt{ArrayN}\ a\ n$ as shorthand for $\texttt{Int}\{v : v < n\} \rightarrow a$. 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 $\forall v.\ v = 4 \Rightarrow v < 10$.

$$\dfrac{\dfrac{\dots}{\Gamma \vdash \texttt{get}\ 10 : \texttt{Int}\{v : v < 10\} \rightarrow \dots} \qquad \dfrac{\dfrac{}{\Gamma \vdash 4 : \texttt{Int}\{v : v = 4\}} \quad \dfrac{\forall v. v = 4 \Rightarrow v < 10}{\Gamma \vdash \texttt{Int}\{v : v = 4\} \preceq \texttt{Int}\{v : v < 10\}}\textsc{S-Base}}{\Gamma \vdash 4 : \texttt{Int}\{v : v < 10\}}\textsc{T-Sub}}{\Gamma \vdash \texttt{get}\ 10\,4 : \texttt{ArrayN}\ a\ 10 \rightarrow a}$$

Importantly, most refinement type systems use syntax-directed rules to destruct subtyping obligations into basic (semantic) implications. For example, in Figure 8 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 $\texttt{get}\ 10\ 4 : \texttt{ArrayN}\ a\ 10 \rightarrow a$ 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 [Vazou et al. 2014a] and Flux [Lehmann et al. 2023]), $\mathcal{L}$ 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 7).

$$\dfrac{\Gamma \vdash e_f : t_f \qquad \Gamma \vdash_w t : k \qquad f : t_f, \Gamma \vdash e : t}{\Gamma \vdash \texttt{let}\ f = e_f\ \texttt{in}\ e : t}\textsc{T-Let}$$

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 [Leino 2010] or Prusti [Astrauskas et al. 2022]) 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 [Knowles and Flanagan 2009] to check dependent function application, *i.e.* the TApp-Exists rule below, instead of the standard type-theoretic TApp-Exact rule.

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

To understand the difference, consider some expression e of type $\texttt{Pos}$ and the identity function f

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

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 $f : x{:}\texttt{Pos} \rightarrow \texttt{Int}\{v{:}v = x\}$. 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 [2009] 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 [Flanagan et al. 1993]. 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 [Cosman and Jhala 2017]. Thus, we choose to use existential types in λ_{RF}.

2.3.3 *Polymorphism.*

Polymorphism is a precious type abstraction [Wadler 1989], 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`	::	$x{:}\texttt{Int} \rightarrow y{:}\texttt{Int} \rightarrow \texttt{Int}\{\nu{:}x \leq \nu \land y \leq \nu\}$
Attempt 2:	*Unrefined Polymorphism*	`max`	::	$x{:}\alpha \rightarrow y{:}\alpha \rightarrow \alpha$
Attempt 3:	*Refined Polymorphism*	`max`	::	$x{:}\alpha \rightarrow y{:}\alpha \rightarrow \alpha\{\nu{:}x \leq \nu \land y \leq \nu\}$
λ_{RF}:	*Kinded Polymorphism*	`max`	::	$\forall \alpha{:}B.x{:}\alpha \rightarrow y{:}\alpha \rightarrow \alpha\{\nu{:}x \leq \nu \land y \leq \nu\}$

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 [Belo et al. 2011], all the types that substitute variables should be refined. For example, as detailed in §6 of [Jhala and Vazou 2021], if a

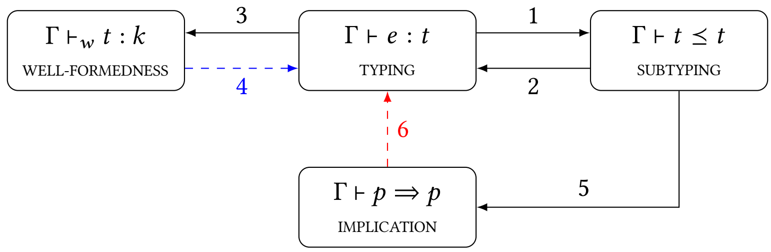

Fig. 2. Dependencies of Typing Judgements in Refinement Types. (Dashed lines do not exist in our formalism.)

type variable is refined with `false` (*i.e.* $\alpha\{v$:`false`$\}$) and gets instantiated with an unrefined function type ($x{:}t_x \rightarrow t$), 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.

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 `error` to denote program errors (such as out-of-bounds indexing of Figure 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 § 4 we inductively define the denotations of each type and Theorem 5.1 formally encodes denotational soundness.

This work, for the first time, mechanizes the soundness of refinement types with semantic subtyping, existential types, and polymorphism. This mechanization was challenging for three main reasons:

Challenge 1: Circularities Figure 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 § 4), 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 8), 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 § 4.4.3 we define implication via type denotations, but as observed by Greenberg [2013], 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 [2013] 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, 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 [Flanagan 2006; Vazou et al. 2018]. 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* [Gordon and Fournet 2010; Rondon et al. 2008]. 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 § 4.4.2) which has never been mechanized.
(3) By *axiomatization* [Lehmann and Tanter 2016]. 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 [Sekiyama et al. 2017]).

Our mechanization follows a combination of the first and the third approach. We specify the interface of implication (via Requirement 2 of § 4.4.1 which is encoded as an inductive data type in the proof mechanization) to articulate the exact properties required by the soundness proof. Then, in § 4.4.3, 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 § 4.4.2 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

Primitives	c	::=	$\texttt{true} \mid \texttt{false} \mid 0,1,2,\ldots \mid \wedge,\neg \mid \leq, c\leq, =, c=$
Values	v	::=	$c \mid x, y, \ldots \mid \lambda x.e \mid \Lambda\alpha{:}k.e$
Terms	e	::=	$v \mid e_1\, e_2 \mid e[t] \mid \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \mid e : t \mid \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \mid \texttt{error}$

Fig. 3. 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	::=	$\texttt{Bool} \mid \texttt{Int} \mid \alpha$	*bool, ints, and type variables*
Types	t	::=	$b\, \{v{:}p\}$	*refined base type*
		|	$x : t_x \rightarrow t$	*function type*
		|	$\exists x{:}t_x.t$	*existential type*
		|	$\forall\alpha{:}k.t$	*polymorphic type*
Environments	Γ	::=	$\varnothing \mid \Gamma, x : t \mid \Gamma, \alpha : k$	*variable and type bindings*

Fig. 4. Syntax of Types. The grey boxes are the extensions to λ_F needed by λ_{RF}. We use τ for λ_F-only types.

to kind checking. Semantic subtyping makes type checking not syntax-directed (thus inversion is not trivial § 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.

3 LANGUAGE

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 [Pierce 2002] 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 § 4.1) and the implication (in type denotations of Figure 9) 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 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.

Kinds & Types Figure 4 shows the syntax of the types, with the grey 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 $b\{v{:}p\}$ comprising a base type b and a *refinement* that

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

$$\frac{}{c\,v \hookrightarrow \delta(c,v)}\textsc{E-Prim} \qquad \frac{}{c[t] \hookrightarrow \delta_T(c,\lfloor t \rfloor)}\textsc{E-TPrim} \qquad \frac{e \hookrightarrow e'}{e:t \hookrightarrow e':t}\textsc{E-PAnn} \qquad \frac{}{v:t \hookrightarrow v}\textsc{E-Ann}$$

$$\frac{e \hookrightarrow e'}{e\,e_1 \hookrightarrow e'\,e_1}\textsc{E-PLApp} \qquad \frac{e \hookrightarrow e'}{v\,e \hookrightarrow v\,e'}\textsc{E-PRApp} \qquad \frac{}{(\lambda x.e)\,v \hookrightarrow e[v/x]}\textsc{E-App} \qquad \frac{}{(\Lambda\alpha{:}k.e)[t] \hookrightarrow e[t/\alpha]}\textsc{E-TApp}$$

$$\frac{e \hookrightarrow e'}{e[t] \hookrightarrow e'[t]}\textsc{E-PTApp} \qquad \frac{e_x \hookrightarrow e'_x}{\texttt{let}\,x = e_x\,\texttt{in}\,e \hookrightarrow \texttt{let}\,x = e'_x\,\texttt{in}\,e}\textsc{E-PLet} \qquad \frac{}{\texttt{let}\,x = v\,\texttt{in}\,e \hookrightarrow e[v/x]}\textsc{E-Let}$$

$$\frac{e \hookrightarrow e'}{\texttt{if}\,e\,\texttt{then}\,e_1\,\texttt{else}\,e_2 \hookrightarrow \texttt{if}\,e'\,\texttt{then}\,e_1\,\texttt{else}\,e_2}\textsc{E-PIf}$$

$$\frac{}{\texttt{if}\,\texttt{true}\,\texttt{then}\,e_1\,\texttt{else}\,e_2 \hookrightarrow e_1}\textsc{E-IfT} \qquad \frac{}{\texttt{if}\,\texttt{false}\,\texttt{then}\,e_1\,\texttt{else}\,e_2 \hookrightarrow e_2}\textsc{E-IfF}$$

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

Fig. 5. The small-step semantics and type substitution.

restricts b to the subset of values v 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 $b\{v{:}\texttt{true}\}$ 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:

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

Environments Figure 4 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 [Aydemir et al. 2005]: 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 § 9). 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 5 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 substitution is defined at the bottom left of Figure 5 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.

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

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. $e\hookrightarrow e'$.*

4 STATIC SEMANTICS

The static semantics of our calculi comprise four main judgment forms: (§ 4.1) *well-formedness* judgments that determine when a type or environment is syntactically well-formed (in λ_F and λ_{RF}); (§ 4.2) *typing* judgments that stipulate that a term has a particular type in a given context (in λ_F and λ_{RF}); (§ 4.3) *subtyping* judgments that establish when one type can be viewed as a subtype of another (in λ_{RF}); and (§ 4.4) *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}.

4.1 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 $\Gamma\vdash_w\tau:k$ 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.

Rules Figure 6 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 $\Gamma\vdash_w\alpha\{x{:}\mathtt{true}\}:B$ 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`. Finally, rule WF-Kind simply states that if

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

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

$$\frac{\Gamma \vdash_w b\{x : \texttt{true}\} : B \quad \forall y \notin \Gamma. y : b, \lfloor\Gamma\rfloor \vdash_F p[y/x] : \texttt{Bool}}{\Gamma \vdash_w b\{x : p\} : B}\text{WF-Refn} \qquad \frac{\Gamma \vdash_w t_x : k_x \quad \forall y \notin \Gamma.\ y : t_x, \Gamma \vdash_w t\ [y/x] : k}{\Gamma \vdash_w x : t_x \rightarrow t : \star}\text{WF-Func}$$

$$\frac{\Gamma \vdash_w t_x : k_x \ \forall y \notin \Gamma. y : t_x, \Gamma \vdash_w t[y/x] : k}{\Gamma \vdash_w \exists x : t_x . t : k}\text{WF-Exis} \qquad \frac{\forall \alpha' \notin \Gamma.\ \alpha' : k, \Gamma \vdash_w t[\alpha'/\alpha] : k_t}{\Gamma \vdash_w \forall \alpha : k . t : \star}\text{WF-Poly}$$

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

$$\frac{}{\vdash_w \varnothing}\text{WFE-Emp} \qquad \frac{\Gamma \vdash_w t_x : k_x \quad \vdash_w \Gamma \quad x \notin \Gamma}{\vdash_w x : t_x, \Gamma}\text{WFE-Bind} \qquad \frac{\vdash_w \Gamma \quad \alpha \notin \Gamma}{\vdash_w \alpha : k, \Gamma}\text{WFE-TBind}$$

Fig. 6. Well-formedness of types and environments. The rules for λ_F exclude the grey boxes.

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.

As for environments, the empty environment is well-formed. A well-formed environment remains well-formed after binding a fresh term or type variable to *resp.* any well-formed type or kind.

4.2 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 7 summarizes the rules that establish typing for both λ_F and λ_{RF}, with grey for the λ_{RF} extensions.

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

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

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 $\lfloor \texttt{ty}(c) \rfloor$. 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 $\texttt{ty}(c)$ 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 1. *(Primitives) For every primitive c,*

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

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

$$\frac{}{\Gamma \vdash c : \mathsf{ty}(c)}\textsc{T-Prim} \qquad \frac{x:t \in \Gamma \quad \Gamma \vdash_w t : k}{\Gamma \vdash x : \mathsf{self}(t, x, k)}\textsc{T-Var} \qquad \frac{\Gamma \vdash e : t \quad \Gamma \vdash_w t : k}{\Gamma \vdash e : t : t}\textsc{T-Ann} \qquad \frac{\Gamma \vdash_w t : k \quad \Gamma \vdash e : s \quad \Gamma \vdash s \preceq t}{\Gamma \vdash e : t}\textsc{T-Sub}$$

$$\frac{\Gamma \vdash e_x : t_x \quad \Gamma \vdash e : x : t_x \to t}{\Gamma \vdash e\, e_x : \exists x{:}t_x.\ t}\textsc{T-App} \qquad \frac{\Gamma \vdash_w t_x : k_x \quad \forall y \notin \Gamma.\, y : t_x, \Gamma \vdash e[y/x] : t\,[y/x]}{\Gamma \vdash \lambda x.e : x : t_x \to t}\textsc{T-Abs} \qquad \frac{\forall \alpha' \notin \Gamma.\ \alpha' : k, \Gamma \vdash e[\alpha'/\alpha] : t[\alpha'/\alpha]}{\Gamma \vdash \Lambda \alpha{:}k.e : \forall \alpha{:}k.t}\textsc{T-TAbs}$$

$$\frac{\Gamma \vdash_w t : k \quad \Gamma \vdash e : \forall \alpha{:}k.s}{\Gamma \vdash e[t] : s[t/\alpha]}\textsc{T-TApp} \qquad \frac{\Gamma \vdash e_x : t_x \quad \Gamma \vdash_w t : k \quad \forall y \notin \Gamma.\, y : t_x, \Gamma \vdash e[y/x] : t\,[y/x]}{\Gamma \vdash \mathtt{let}\, x = e_x\, \mathtt{in}\, e : t}\textsc{T-Let}$$

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

Fig. 7. Typing rules. The judgment $\Gamma \vdash_F e : \tau$ is defined by excluding the grey boxes.

Theorem 3 of [Vazou et al. 2014b] proves that a terminating `fix` constant satisfies requirement 1.

To type constants, rule T-Prim gives the type $ty(c)$ to any built-in primitive c, in any context.

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 [Ou et al. 2004] $\mathsf{self}(t, x, k)$ provided that $\Gamma \vdash_w t : k$. This rule is crucial in practice, to enable path-sensitive "occurrence" typing [Tobin-Hochstadt and Felleisen 2008], 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.

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

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 [2009]. 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 [Flanagan et al. 1993]), but our metatheory would be more complex since ANF is not preserved under the small step operational semantics.

Other Typing Rules 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 $e[t]$.

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

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

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

$$\frac{\forall \alpha' \notin \Gamma. \quad \alpha' : k, \Gamma \vdash t_1[\alpha'/\alpha] \preceq t_2[\alpha'/\alpha]}{\Gamma \vdash \forall \alpha{:}k.t_1 \preceq \forall \alpha{:}k.t_2} \text{ S-Poly} \qquad \frac{\forall y \notin \Gamma. \quad y : b, \Gamma \vdash p_1[y/x] \Rightarrow p_2[y/x]}{\Gamma \vdash b\{x{:}p_1\} \preceq b\{x{:}p_2\}} \text{ S-Base}$$

Fig. 8. Subtyping Rules.

For the λ_F variant of T-TApp, we erase the refinements (via $\lfloor t \rfloor$) before checking well-formedness and performing the substitution. 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.

4.3 Subtyping

The *subtyping* judgment $\Gamma \vdash s \preceq t$, defined in Figure 8, 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 7).

Subtyping Rules Rules S-Bind and S-Wit establish subtyping for existential types [Knowles and Flanagan 2009], *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 $\exists x{:}t_x.t'$.

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.

4.4 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 [Swamy et al. 2016; Vazou et al. 2014a], 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 (§ 4.4.1) that precisely captures all the requirements that the implication judgement should satisfy to establish the soundness of λ_{RF}. Then, we define two alternative implementations of the interface:

a logical implementation (§ 4.4.2) that is used in refinement type implementations and a denotational implementation (§ 4.4.3) that we used to complete our mechanized proof.

4.4.1 Implication's Interface. In our mechanization, following Lehmann and Tanter [2016], we encode implication as an axiomatized judgment that satisfies the requirements below.

Requirement 2 (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 1 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 [Knowles and Flanagan 2009]. Our Strengthening and Subst II cases are required for polymorphism, thus they do not appear in Knowles and Flanagan [2009]'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 § 4.4.3 we define (and mechanize) an implementation, *i.e.* inhabitant, of the interface thus show our assumptions are consistent.

4.4.2 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:

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

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 [de Moura and Bjørner 2008], while the soundness of this implementation (concretely statement 7 of Requirement 2) is hinted by Theorem 2 of [Vazou et al. 2014b]. Chen [2022] 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.

4.4.3 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.

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

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

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

Fig. 9. Denotations of Types and Environments.

The refinements p_1 and p_2 are boolean expressions, so evaluation uses the operational semantics of Figure 5. 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 9 defines the denotations of types and environments. Following Flanagan [2006], 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 9, 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). 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 (§ 8) that the denotational implementations satisfies the statements of Requirement 2.

5 SOUNDNESS

Our development for λ_F (§ 5) follows the standard presentation of System F's metatheory by Pierce [2002]. The main difference is that ours includes well-formedness of types and environments, which help with mechanization [Rémy 2021] and are crucial in λ_{RF} when formalizing refinements.

Figure 10 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*

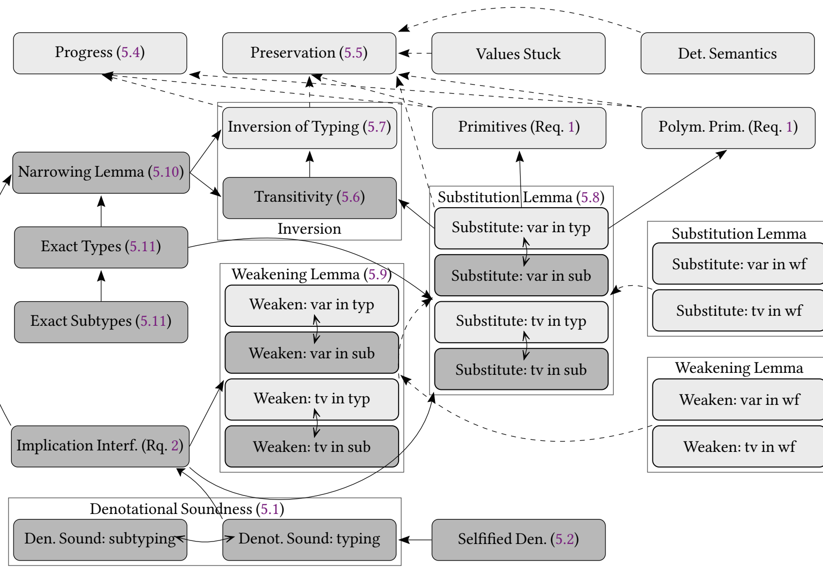

Fig. 10. Dependencies in the metatheory. We write "var" and "tv" to resp. abbreviate term and type variables.

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 1).

5.1 Denotational Soundness

Denotational soundness connects syntactic typing and subtyping with the type denotations (of Figure 9). 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 $v \in [\![b\{x : p \land x = v\}]\!]$ as $(p \land x = v)[v/x]$ certainly evaluates to `true`.

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} *and* λ_F*)*

(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 $\varnothing \vdash e : \tau$, 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 1, 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 $\varnothing \vdash e : \tau$ and $e \hookrightarrow e'$, then $\varnothing \vdash e' : \tau$.*

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 T-APP and T-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 10 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. [2005], 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 $\Gamma \vdash (\lambda w.e) : x{:}t_x \rightarrow t$ 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 $\Gamma \vdash (\lambda w.e) : x{:}t_x \rightarrow t$, 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 $\Gamma \vdash t_1 \preceq x{:}t_x \rightarrow t$. 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 $w{:}s_w \rightarrow s$ such that $\Gamma \vdash (\lambda w.e) : w{:}s_w \rightarrow s$ and $\Gamma \vdash w{:}s_w \rightarrow s \preceq x{:}t_x \rightarrow t$ (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 [Aydemir et al. 2008] 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 both subtyping and typing is preserved after selfification.

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* $\Gamma \vdash v:\mathrm{self}(t,v,k)$.

6 LIQUIDHASKELL & REFINED DATA PROPOSITIONS

In § 7 we 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.

6.1 LiquidHaskell

LiquidHaskell's core proof system is λ_{RF}, that is, it is using the typing judgement presented in fig. 7 to check if a Haskell program satisfies its refinement type annotations. The expression language checked by LiquidHaskell is GHC's intermediate language (CoreSyn [Sulzmann et al. 2007]) which is a superset of λ_{RF} that also includes literals, datatypes, and coercions. Thus, LiquidHaskell's typing judgement is extended to include these constructs. To guess the unknown types of fig. 7 (*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 [Rondon et al. 2008]. To check the implications LiquidHaskell uses the I-Log rule of § 4.4.2 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 [Vazou et al. 2018] 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 to permit constructive reasoning about potentially non-terminating properties, as required for meta-theoretic proofs.

Refined data propositions encode 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` *resp.* represent the *propositions* $\gamma \vdash e : t$ and $\gamma \vdash s \preceq t$.

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 7, 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 7.

```
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 $\gamma \vdash e : t$".

Implementation of Data Propositions Data propositions are a novel feature required to encode inductive propositions in the mechanization of λ_{RF}. (Parker et al. [2019] developed a LiquidHaskell metatheoretic proof but before data propositions and thus had to axiomatize a terminating evaluation relation; see § 9.) 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 [Jhala and Vazou 2021]. 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.

7 LIQUIDHASKELL MECHANIZATION

We mechanized type safety (Theorem 5.3) of λ_{RF} in both Coq 8.15.1 and LiquidHaskell 8.10.7.1 (submitted as anonymous supplementary material). In LiquidHaskell we use refined data propositions (§ 6) to specify the static (*e.g.* typing, subtyping, well-formedness) and dynamic (*i.e.* small-step transitions and their closure) semantics of λ_{RF}. 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 (§7.3). 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.

The LiquidHaskell mechanization is simplified by SMT-automation (§ 7.1) and consists of proofs implemented as recursive functions that construct evidence to establish propositions by induction

(§ 7.2). Note that while Haskell types are inhabited by diverging ⊥ 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 SMT Solvers, Arithmetic, and Set Theory

The most tedious part in 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.

Intrinsic Verification LiquidHaskell embeds the functions of the standard `Data.Set` Haskell library as SMT set operators. Given a Haskell function, *e.g.* the set of free variables in an expression, this embedding, combined with SMT's support for set theory, lets LiquidHaskell prove properties about free variables "for free". For example, 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.* $\mathsf{fv}(e[v_x/x]) \subseteq \mathsf{fv}(v_x) \cup (\mathsf{fv}(e) \setminus \{x\})$. 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*.

7.2 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 with a `Step` data proposition that encodes Figure 5 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.

7.3 Quantitative Results

We provide a mechanically checked proof of the type safety in § 5, that only assumes the requirements 1 and 2. Concretely, we assumed the primitives Requirement 1 for some constants of λ_{RF} because it was too strenuous to mechanically prove without interactive aid. In LiquidHaskell type denotations (of Figure 9) cannot be currently encoded: since they include ∀-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 (§ 4.4.3) 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 [Aydemir et al. 2005]. 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* [Aydemir et al. 2008; Pollack 1993], where free variables are named, but bound variables are represented by deBruijn 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 5) 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.

Table 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 12,800 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 (§ 8) 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.

8 COQ MECHANIZATION

Our Coq mechanization proves both type safety and denotation soundness, *i.e.* all the statements of § 5 and serves as a comparison for the metatheoretical development abilities of the two theorem provers. In Coq, Req. 1 is proved (using Coq's interactive development) and type denotations (of Figure 9) are defined as recursive functions using Equations [Sozeau and Mangin 2019], which make both the definition the denotational implementation of the implication (§ 4.4.3) and the proof the denotational soundness (Theorem 5.1) possible. 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. [2017] previously compared LiquidHaskell and Coq as theorem

	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

Table 1. Quantitative mechanization details. We split each development into sets of modules pertaining to regions of Figure 10 and for each we count lines of specification (definitions, lemma statements) and of proof.

provers, but their mechanizations were an order of magnitude smaller than ours and did not use data propositions (§ 6), which permit constructive LiquidHaskell proofs.

Coq vs. LiquidHaskell Coq has a tiny TCB and strong foundational mechanized soundness guarantees [Sozeau et al. 2020]. 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.

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 [Coquand and Paulin 1990]. 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 [Pierce et al. 2022]. 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 [Sozeau and Mangin 2019] 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 (thm. 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.
```

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. 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 hypothesis. 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. 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. 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.

9 RELATED WORK

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

Hybrid & Contract Systems Flanagan [2006] formalizes on paper a monomorphic lambda calculus with refinement types that differs from our λ_{RF} in two ways. First, in Flanagan [2006]'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. [2017] 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 $\text{div} :: \texttt{Int} \to \texttt{Int}\{n{:}n \neq 0\} \to \texttt{Int}$ 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. [2017] is highly simplified. Static refinement type systems (as summarized by Jhala and Vazou [2021]) 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. [2002] use the set theoretic models of types to decide subtyping. Castagna and Frisch [2005] present an algorithm that decides semantic subtyping for a core calculus with functional types. Like λ_{RF}, Castagna and Frisch [2005] 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 than 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 [2016]'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. [2019] 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` → `Pos` is subtype of `Pos` → `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 [2022] 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 [2023] 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` [Parker et al. 2019] 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.

10 CONCLUSIONS & FUTURE WORK

We presented and formalized, for the first time, the soundness of λ_{RF}, a refinement calculus with semantic subtyping, existential types, and parametric polymorphism, which are critical for practical refinement typing. Our metatheory is mechanized in both Coq and 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 [Norell 2007], Isabelle [Nipkow et al. 2002], Beluga [Pientka 2010], Dafny [Leino 2010], or F* [Martínez et al. 2019], 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. 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. The immediate next step is to extend the language to include literals, casts, and 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 [Necula 1997] 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 [Norell 2007] and envisioned by Redmond et al. [2021]. 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.

11 DATA AVAILABILITY STATEMENT

The source code for our mechanizations in Coq and LiquidHaskell, together with instructions on how to replicate the results, are available on Zenodo [Borkowski et al. 2023a]. Additionally, a virtual appliance for Oracle VM VirtualBox is available on Zendo [Borkowski et al. 2023b] to assist with replication.

ACKNOWLEDGMENTS

We thank James Parker for a helpful discussion about data propositions and the anonymous reviewers for the useful comments and suggestions. This work was supported by the NSF grants CNS-2120642, CNS-2155235, CCF-1918573, CCF-1911213, CCF-1955457, the Horizon Europe ERC Starting Grant CRETE (GA: 101039196), the US Office of Naval Research HACKCRYPT (Ref. N00014-19-1-2292), and generous gifts from Microsoft Research.

REFERENCES

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

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3603)*, Joe Hurd and Thomas F. Melham (Eds.). Springer, 50–65. https://doi.org/10.1007/11541868_4

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

João Filipe Belo, Michael Greenberg, Atsushi Igarashi, and Benjamin C. Pierce. 2011. Polymorphic Contracts. In *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 (Lecture Notes in Computer Science, Vol. 6602)*, Gilles Barthe (Ed.). Springer, 18–37. https://doi.org/10.1007/978-3-642-19718-5_2

Michael H. Borkowski, Niki Vazou, and Ranjit Jhala. 2023a. *Artifact for "Mechanizing Refinement Types"*. https://doi.org/10.5281/zenodo.8425960

Michael H. Borkowski, Niki Vazou, and Ranjit Jhala. 2023b. *Artifact Virtual Machine for "Mechanizing Refinement Types"*. https://doi.org/10.5281/zenodo.8425176

Giuseppe Castagna and Alain Frisch. 2005. A Gentle Introduction to Semantic Subtyping. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Lisbon, Portugal) *(PPDP '05)*. Association for Computing Machinery, New York, NY, USA, 198–208. https://doi.org/10.1145/1069774.1069793

Zilin Chen. 2022. A Hoare Logic Style Refinement Types Formalisation. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development* (Ljubljana, Slovenia) *(TyDe 2022)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3546196.3550162

Thierry Coquand and Christine Paulin. 1990. Inductively Defined Types. In *COLOG-88*, Per Martin-Löf and Grigori Mints (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 50–66. https://doi.org/10.1007/3-540-52335-9_47

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

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Cormac Flanagan. 2006. Hybrid Type Checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '06)*. Association for Computing Machinery, New York, NY, USA, 245–256. https://doi.org/10.1145/1111037.1111059

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 237–247. https://doi.org/10.1145/155090.155113

Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. 2011. Modular Code-Based Cryptographic Verification. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '11)*. Association for Computing Machinery, New York, NY, USA, 341–350. https://doi.org/10.1145/2046707.2046746

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

Jad Elkhaleq Ghalayini and Neel Krishnaswami. 2023. Explicit Refinement Types. *Proc. ACM Program. Lang.* 7, ICFP, Article 195 (aug 2023), 28 pages. https://doi.org/10.1145/3607837

Andrew D. Gordon and C. Fournet. 2010. Principles and Applications of Refinement Types. In *Logics and Languages for Reliability and Security*. IOS Press. https://doi.org/10.3233/978-1-60750-100-8-73

Michael Greenberg. 2013. *Manifest Contracts*. Ph.D. Dissertation. University of Pennsylvania. https://repository.upenn.edu/edissertations/468/

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

Ranjit Jhala and Niki Vazou. 2021. Refinement Types: A Tutorial. *Found. Trends Program. Lang.* 6, 3-4 (2021), 159–317. https://doi.org/10.1561/2500000032

Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. 2016. Occurrence Typing modulo Theories. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 296–309. https://doi.org/10.1145/2908080.2908091

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

Kenneth Knowles and Cormac Flanagan. 2009. Compositional Reasoning and Decidable Checking for Dependent Contract Types. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification* (Savannah, GA, USA) *(PLPV '09)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/1481848.1481853

Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. *Proc. ACM Program. Lang.* 7, PLDI, Article 169 (jun 2023), 25 pages. https://doi.org/10.1145/3591283

Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. 2021. STORM: Refinement Types for Secure Web Applications. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 441–459. https://www.usenix.org/conference/osdi21/presentation/lehmann

Nico Lehmann and Éric Tanter. 2016. Formalizing Simple Refinement Types in Coq. In *2nd International Workshop on Coq for Programming Languages (CoqPL'16)*. St. Petersburg, FL, USA.

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

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. 2019. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-030-17184-1_2

George C. Necula. 1997. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 106–119. https://doi.org/10.1145/263699.263712

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. https://link.springer.com/book/10.1007/3-540-45949-9

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Chalmers.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. Dynamic Typing with Dependent Types. In *Exploring New Frontiers of Theoretical Informatics*, Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell (Eds.). Springer US, Boston, MA, 437–450. https://doi.org/10.1007/1-4020-8141-3_34

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

Brigitte Pientka. 2010. Beluga: Programming with Dependent Types, Contextual Data, and Contexts. In *Functional and Logic Programming*, Matthias Blume, Naoki Kobayashi, and Germán Vidal (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–12. https://doi.org/10.1007/978-3-642-12251-4_1

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

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

Randy Pollack. 1993. Closure Under Alpha-Conversion. In *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers (Lecture Notes in Computer Science, Vol. 806)*, Henk Barendregt and Tobias Nipkow (Eds.). Springer, 313–332. https://doi.org/10.1007/3-540-58085-9_82

Patrick Redmond, Gan Shen, and Lindsey Kuper. 2021. Toward Hole-Driven Development with Liquid Haskell. *CoRR* abs/2110.04461 (2021). arXiv:2110.04461 https://arxiv.org/abs/2110.04461

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

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

Taro Sekiyama, Atsushi Igarashi, and Michael Greenberg. 2017. Polymorphic Manifest Contracts, Revised and Resolved. *ACM Trans. Program. Lang. Syst.* 39, 1 (2017), 3:1–3:36. https://doi.org/10.1145/2994594

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

Matthieu Sozeau and Cyprien Mangin. 2019. Equations Reloaded: High-Level Dependently-Typed Functional Programming and Proving in Coq. *Proc. ACM Program. Lang.* 3, ICFP, Article 86 (jul 2019), 29 pages. https://doi.org/10.1145/3341690

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (Nice, Nice, France) *(TLDI '07)*. Association for Computing Machinery, New York, NY, USA, 53–66. https://doi.org/10.1145/1190315.1190324

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. 2016. Dependent Types and Multi-Monadic Effects in F*. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2837614.2837655

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 395–406. https://doi.org/10.1145/1328438.1328486

Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. 2017. A Tale of Two Provers: Verifying Monoidal String Matching in Liquid Haskell and Coq. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell* (Oxford, UK) *(Haskell 2017)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/3122955.3122963

Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014a. LiquidHaskell: Experience with Refinement Types in the Real World. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell* (Gothenburg, Sweden) *(Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 39–51. https://doi.org/10.1145/2633357.2633366

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014b. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

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

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (Imperial College, London, United Kingdom) *(FPCA '89)*. Association for Computing Machinery, New York, NY, USA, 347–359. https://doi.org/10.1145/99370.99404

Received 2023-07-11; accepted 2023-11-07