# Mechanizing Refinement Types

MICHAEL 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 meta-theoretic accounting of the *soundness* of refinement type systems using this combination has proved elusive. We present $\lambda_{RF}$, 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 full 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.

## 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. Refinements on types have been successfully used to define sophisticated concepts (*e.g.* secrecy [Fournet et al. 2011], resource constraints [Knoth et al. 2020], security policies [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 troubling the programmer for explicit type casts. For example, consider a positive expression $e : \texttt{Pos}$ and a function expecting natural numbers $f : \texttt{Nat} \rightarrow \texttt{Int}$. To type check the application $f\ e$, the refinement type system will implicitly convert the type of $e$ from Pos to Nat, because $0 < v \Rightarrow 0 \leq v$ semantically holds. 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 identity function $g : a \rightarrow a$.

As is often the case with useful ideas, the engineering of practical tools has galloped far ahead of the development of the meta-theoretical foundations for refinements with subtyping and polymorphism. In fact, semantic subtyping is very tricky as it is mutually defined with typing, leading to metatheoretic proofs with circular dependencies. 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. 2011a].

In this paper we formalize $\lambda_{RF}$, a core calculus with a refinement type system that combines semantic subtyping with polymorphism. Our development has four concrete contributions.

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

***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, $\lambda_{RF}$ 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 2009b] 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 $\lambda_{RF}$ by proving soundness, which says that if $e$ has a type then, either $e$ is a value or it can step to another term of the same type. The combination of semantic subtyping, polymorphism, and existentials makes the soundness proof challenging with circular dependencies that do not arise in standard (unrefined) calculi. To ease the presentation and tease out the essential ingredients of the proof we stage the metatheory. First, we review an unrefined *base* language $\lambda_F$, a classic System F [Pierce 2002] with primitive `Int` and `Bool` types (§ 5). Next, we show how refinements (kinds, subtyping, and existentials) must be accounted for to establish the soundness of $\lambda_{RF}$ (§ 6).

***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 (§ 7). Without data propositions reasoning about potentially non-terminating computations was not possible in LiquidHaskell, thereby precluding even simple meta-theoretic developments such as the soundness of $\lambda_F$ let alone $\lambda_{RF}$.

***4. Mechanization*** Our final contribution is to fully mechanize the metatheory of $\lambda_{RF}$ *twice*: using LiquidHaskell and Coq. We formalized $\lambda_{RF}$ in LiquidHaskell (§ 8) to evaluate the feasibility of such substantial meta-theoretical 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 (§ 9) to compare the two alternatives. The Coq development is slightly shorter (about 7,800 lines), much faster (about 30 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, $\lambda_{RF}$ lays the foundation for the mechanized metatheory of LiquidHaskell.

## 2 OVERVIEW

Our overall strategy is to present the metatheory for $\lambda_{RF}$ in two parts. First, we review the metatheory for $\lambda_F$: a familiar starting point that corresponds to the full language with refinements erased (§ 5). Second, we use the scaffolding established by $\lambda_F$ to highlight the extensions needed to develop the metatheory for refinements in $\lambda_{RF}$ (§ 6). Let's begin with a high-level overview that describes a proof skeleton that is shared across the developments for $\lambda_F$ and $\lambda_{RF}$, the specific challenges posed by refinements, and the machinery needed to go from the simpler $\lambda_F$ to the refined $\lambda_{RF}$.

***Types and Terms*** Both $\lambda_F$ and $\lambda_{RF}$ have the same syntax for *terms* $e$ (Fig. 2). $\lambda_F$ has the usual syntax for *types* $t$ familiar from System F, while $\lambda_{RF}$ additionally allows ($\lambda_F$'s) types to be *refined* by terms (respectively, the white parts and all of Fig. 3), and existential types. Both languages include a notion of *kinds* $k$ that qualify the types that are allowed to be refined.

***Judgments*** Both languages have *typing* judgments $\Gamma \vdash e : t$ which say that a term $e$ has type $t$ with respect to a binding environment (*i.e.* context) $\Gamma$. Additionally, both languages have *well-formedness* judgments $\Gamma \vdash_w t : k$ which say that a type $t$ has the kind $k$ in context $\Gamma$, by requiring that the free variables in $t$ are appropriately bound in the environment $\Gamma$. (Some presentations of $\lambda_F$ [Aydemir

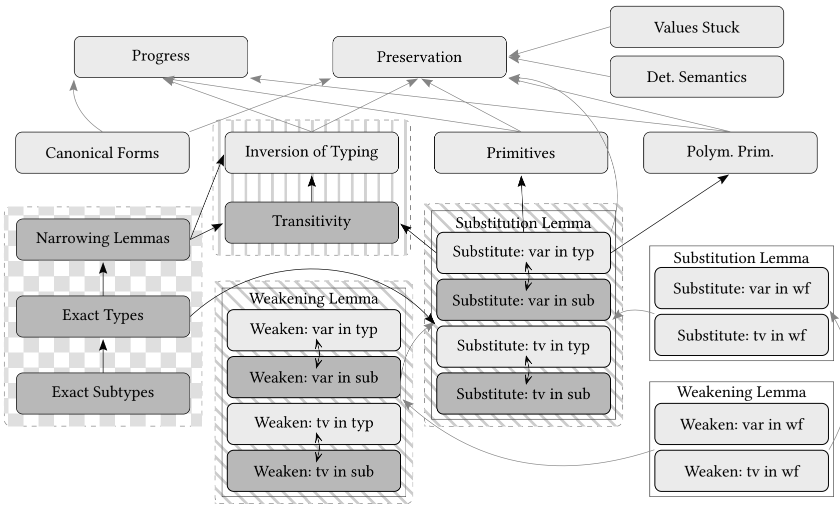

Fig. 1. Dependencies in the metatheory. We write "var" and "tv" to resp. abbreviate term and type variables.

et al. 2008] require well-formedness judgments to mechanize metatheory). Crucially, $\lambda_{RF}$ has a *subtyping* judgment $\Gamma \vdash t_1 \preceq t_2$ which says that type $t_1$ is a subtype of $t_2$ in context $\Gamma$. Subtyping for refined base types is established via an axiomatized *implication* judgment $\Gamma \vdash p \Rightarrow q$ which says that the term $p$ logically implies the term $q$ whenever their free variables are given values described by $\Gamma$. To prove soundness, we *axiomatized* the properties of the implication checking oracle.

***Proof Landscape*** Fig. 1 charts the overall landscape of our formal development as a dependency graph of the main lemmas which establish meta-theoretic properties of the different judgments. Nodes shaded light grey represent lemmas in the metatheories for $\lambda_F$ and $\lambda_{RF}$. The dark grey nodes denote lemmas that only appear in $\lambda_{RF}$. An arrow shows a dependency: the lemma at the *tail* is used in the proof of the lemma at the *head*. Darker arrows are dependencies in $\lambda_{RF}$ only.

***Soundness via Preservation and Progress*** For both $\lambda_{RF}$ and $\lambda_F$ we establish soundness via

- ***Progress:*** If a closed term is well-typed, then either it is a value or it can be further evaluated;
- ***Preservation:*** If a closed term is well-typed, then its type is preserved under evaluation.

The type soundness theorem states that a well-typed closed term cannot become *stuck*: any sequence of evaluation steps will either end with a value or the sequence can be extended by another step. Next, we describe the lemmas used to establish preservation and progress for $\lambda_F$ and then outline the essential new ingredients that demonstrate soundness for the refined $\lambda_{RF}$.

## 2.1 Metatheory for $\lambda_F$

***Progress*** in $\lambda_F$ is standard as the typing rules are syntax-directed. The top-level rule used to obtain the typing derivation for a term $e$ uniquely determines the syntactic structure of $e$ which lets us use the appropriate small-step reduction rule to obtain the next step of the evaluation of $e$.

***Preservation*** says that when a well-typed expression $e$ steps to $e'$, then $e'$ is also well-typed. As usual, the non-trivial case is when the step is a type abstraction $\Lambda\alpha{:}k.e$ (respectively lambda abstraction $\lambda x.e$) *applied to* a type (respectively value), in which case the term $e'$ is obtained by substituting the type or value appropriately in $e$. Thus, our $\lambda_F$ metatheory requires us to prove a *Substitution Lemma*, which describes how typing judgments behave under substitution of free term or type variables. Additionally, some of our typing rules use well-formedness judgments and so we must also prove that well-formedness is preserved by substitution.

***Substitution*** requires some technical lemmas that let us weaken judgments by adding any fresh variable to the binding environment.

***Primitives*** Finally, the primitive reduction steps (*e.g.* arithmetic operations) require the assumption that the reduction rules defined for the built-in primitives are type preserving.

## 2.2 What's hard about Refinements?

***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, consider a function $f$ that requires natural numbers as input, applied to a positive argument $e$. Let $\Gamma \doteq f : \texttt{Nat} \to \texttt{Int}, e : \texttt{Pos}$. The application $f\ e$ 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 positive integers are always naturals by checking the validity of the formula $\forall v.\ 0 < v \Rightarrow 0 \leq v$.

$$\dfrac{\dfrac{}{\Gamma \vdash f : \texttt{Nat} \to \texttt{Int}}\textsc{T-Var} \qquad \dfrac{\dfrac{}{\Gamma \vdash e : \texttt{Pos}}\textsc{T-Var} \qquad \dfrac{\forall v.\ 0 < v \Rightarrow 0 \leq v}{\Gamma \vdash \texttt{Pos} \leq \texttt{Nat}}\textsc{S-Base}}{\Gamma \vdash e : \texttt{Nat}}\textsc{T-Sub}}{\Gamma \vdash f\ e : \texttt{Int}}\textsc{T-App}$$

Importantly, most refinement type systems use type-constructor directed rules to destruct subtyping obligations into basic (semantic) implications. For example, in Fig. 7 the rule S-Fun states that functions are covariant on the result and contravariant on the arguments. Thus, a refinement type system can, without any annotations or casts, decide that $e : \texttt{Nat} \to \texttt{Pos}$ is a suitable argument for the higher order function $f : (\texttt{Pos} \to \texttt{Nat}) \to \texttt{Int}$.

***Existentials*** For compositional and decidable type checking, some refinement type systems use an existential type [Knowles and Flanagan 2009a] to check dependent function application, *i.e.* the TApp-Exists rule below, instead of the standard type-theoretic TApp-Exact rule.

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

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

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

The application $f\ e$ is typed as $\texttt{Int}\{v : v = e\}$ with the TApp-Exact rule, which has two problems. First, the information that $e$ is positive is lost. To regain this information the system needs to re-analyze the expression $e$ breaking compositional reasoning. Second, the arbitrary expression $e$ enters the refinement logic making it impossible for the system to restrict refinements into decidable logical fragments. Using the TApp-Exists rule, both of these problems are addressed. 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 [2009a] 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 `let` $x = e_1$ `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 $\lambda_{RF}$.

***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` | $=$ | $\lambda x\, y.\mathtt{if}\ x < y\ \mathtt{then}\ y\ \mathtt{else}\ x$ |
| Attempt 1: | *Monomorphism* | `max` | $::$ | $x{:}\mathtt{Int} \to y{:}\mathtt{Int} \to \mathtt{Int}\{v : x \leq v \land y \leq v\}$ |
| Attempt 2: | *Unrefined Polymorphism* | `max` | $::$ | $x{:}\alpha \to y{:}\alpha \to \alpha$ |
| Attempt 3: | *Refined Polymorphism* | `max` | $::$ | $x{:}\alpha \to y{:}\alpha \to \alpha\{v : x \leq v \land y \leq v\}$ |
| $\lambda_{RF}$: | *Kinded Polymorphism* | `max` | $::$ | $\forall\, \alpha{:}B.\, x{:}\alpha \to y{:}\alpha \to \alpha\{v : x \leq v \land y \leq v\}$ |

As a first attempt, we give `max` a monomorphic type, stating that the result of `max` is an integer greater or equal to any 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 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 between any values (including functions). Second, if refinements on type variables are allowed, then, for soundness [Belo et al. 2011b], all the types that substitute variables should be refined. For example, if a type variable is refined with `false` (that is, $\alpha\{v : \mathtt{false}\}$) and gets instantiated with an unrefined function type ($x{:}t_x \to t$), then the `false` refinement is lost and the system becomes unsound.

***Base Kind when Refined*** To preserve the benefits on 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 with the ones that cannot. 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 could be implemented via typeclass constraints, if our system contained data types.

## 2.3 From $\lambda_F$ to $\lambda_{RF}$

The metatheory for $\lambda_{RF}$ requires us to enrich that of $\lambda_F$ with three essential and non-trivial blocks — shown as shaded regions in Fig. 1 — that help surmount the challenges posed by the combination of refinements with existentials, subtyping and polymorphism.

***Typing Inversion*** First, thanks to (refinement) subtyping $\lambda_{RF}$ is not syntax directed, and so we cannot just get derivations of subterms by inversion. For example, we cannot directly invert a derivation $\Gamma \vdash \lambda x.e : x{:}t_x \to t$ to obtain a typing derivation that the body $e$ has type $t$ because the above derivation may have been established using (multiple instances of) subtyping. The typing inversion lemmas addresses this problem by using the *transitivity of subtyping* to restructure the judgment tree to collapse all use of subtyping in a way that lets us invert the non-subtyping judgment to conclude that if a term (*e.g.* $\lambda x.e$) is well-typed, then its components (*e.g.* $e$) are also well-typed. The proof of transitivity of subtyping is non-trivial due to the presence of existential

$$\begin{array}{rcll}
\textbf{\textit{Primitives}} & c & ::= & \texttt{true} \mid \texttt{false} \mid 0,1,2,\dots \mid \wedge, \neg \mid \leq, c\!\leq, =, c\!= \\
\textbf{\textit{Values}} & v & ::= & c \mid x, y, \dots \mid \lambda x.e \mid \Lambda \alpha{:}k.e \\
\textbf{\textit{Terms}} & e & ::= & v \mid e_1\, e_2 \mid e[t] \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid e : t
\end{array}$$

Fig. 2. Syntax of Primitives, Values, and Expressions.

$$\begin{array}{rclll}
\textbf{\textit{Kinds}} & k & ::= & B \mid \star & \textit{base and star kind} \\
\textbf{\textit{Predicates}} & p & ::= & \{e \mid \exists \Gamma.\, \Gamma \vdash_F e : \texttt{Bool}\} & \textit{boolean-typed terms} \\
\textbf{\textit{Base Types}} & b & ::= & \texttt{Bool} \mid \texttt{Int} \mid \alpha & \textit{booleans, integers, and type variables} \\
\textbf{\textit{Types}} & t & ::= & b\{v : p\} & \textit{refined base type} \\
 & & \mid & x{:}t_x \to t & \textit{function type} \\
 & & \mid & \exists\, x{:}t_x.\, t & \textit{existential type} \\
 & & \mid & \forall\, \alpha{:}k.\, t & \textit{polymorphic type} \\
\textbf{\textit{Environments}} & \Gamma & ::= & \varnothing \mid \Gamma, x{:}t \mid \Gamma, \alpha{:}k & \textit{variable and type bindings}
\end{array}$$

Fig. 3. Syntax of Types. The grey boxes are the extensions to $\lambda_F$ needed by $\lambda_{RF}$. We use $\tau$ for $\lambda_F$-only types.

types. We cannot proceed by structural induction on the two subtyping judgments ($\Gamma \vdash t_1 \preceq t_2$ and $\Gamma \vdash t_2 \preceq t_3$), because the subderivations need to be transformed by substitution and narrowing, operations that could increase their size (§ 6.1). Instead, our proof goes by induction on a more intricate size (the combined depth of types $t_1$, $t_2$, and $t_3$).

***Subtyping*** The critical difference between the two metatheories is that $\lambda_{RF}$ has implicit subtyping that introduces a mutual dependency between lemmas for typing and subtyping judgments. Typing depends on subtyping due to the subsumption rule (T-Sub, Fig. 6) that lets us weaken the type of a term with a super-type. Conversely, subtyping depends on typing because of the rule (S-Witn, Fig. 7) which establishes subtyping between *existential* types. Thanks to this mutual dependency, all of the lemmas from $\lambda_F$ that relate to typing judgments, *i.e.* weakening and substitution, are now mutually recursive with versions for subtyping (see the diagonal lined area in Fig. 1).

***Narrowing*** Finally, due to subtyping, the proofs of the inversion and substitution lemmas for $\lambda_{RF}$ require *narrowing* lemmas that allow us to replace a type inside the binding environment of a judgment with a subtype, thus "narrowing" the scope of the judgment. Due to the mutual dependencies between the typing and subtyping judgments, we must prove narrowing for both judgments. A few key cases of these proofs require other technical lemmas shown in the checkerboard region of Fig. 1. For example, the case for the "occurrence-typing" rule T-Var, that crucially enables path-sensitive reasoning, uses the "extact types" lemma that *selfifies* [Ou et al. 2004] the types in the judgments.

## 3 LANGUAGE

To cut the circularities in the metatheory, we formalize refinements using two calculi. The first is the *base* language $\lambda_F$: 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 $\lambda_{RF}$ which extends $\lambda_F$ 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 typing judgments be mutually dependent. We use the grey highlights for the extensions to $\lambda_F$ required for $\lambda_{RF}$.

### 3.1 Syntax

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

***Constants, Values and Terms*** Fig. 2 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 $\lambda$- and type- abstractions. Finally, the *terms* $e$ comprise values, value- and type- applications, let-binders and annotated expressions.

***Kinds & Types*** Fig. 3 shows the syntax of the types, with the grey boxes indicating the extensions to $\lambda_F$ required by $\lambda_{RF}$. In $\lambda_{RF}$, only base types `Bool` and `Int` can be refined: we do not permit refinements for functions and polymorphic types. $\lambda_{RF}$ 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 $\alpha$. The simplest type is of the form $b\{v : p\}$ comprising a base type $b$ and a *refinement* that 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, $\lambda_F$ version of a refined $\lambda_{RF}$ 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*** Fig. 3 describes the syntax of typing environments $\Gamma$ 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 § 10). 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

Fig. 4 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 $\delta$ and $\delta_T$.

***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 Fig. 4 and it is standard except for the last line which defines the substitution of a type variable $\alpha$ in a refined type variable $\alpha\{x : p\}$ with a (potentially refined) type $t_\alpha$. To do this substitution, we combine $p$ with the type $t_\alpha$ by using $\texttt{refine}(t_\alpha, p, x)$ which essentially conjoins the refinement $p$ to the top-level refinement of a base-kinded $t_\alpha$. 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

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

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

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

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

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

Fig. 4. The small-step semantics and type substitution.

$\delta_T(c, \lfloor t \rfloor)$ specifies the reduction rules for type application on the polymorphic built-in primitives.

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

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

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

## 4 STATIC SEMANTICS

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

### 4.1 Well-formedness

***Judgments*** The judgment $\Gamma \vdash_w t : k$ says that the type $t$ is well-formed in the environment $\Gamma$ and has kind $k$. The judgment $\vdash_w \Gamma$ says that the environment $\Gamma$ is well formed, meaning that it only binds to well-formed types. Well-formedness is also used in the (unrefined) system $\lambda_F$, where $\Gamma \vdash_w \tau : k$ means that the (unrefined) $\lambda_F$ type $\tau$ is well-formed in environment $\Gamma$ and has kind $k$ and $\vdash_w \Gamma$ means that the free type variables of the environment $\Gamma$ are bound earlier in the environment.

***Rules*** Fig. 5 summarizes the rules that establish the well-formedness of types and environments. Rule WF-Base states that the two closed base types (`Int` and `Bool`, refined with `true` in $\lambda_{RF}$) are well-formed and have base kind. Similarly, rule WF-Var says that a type variable $\alpha$ is well-formed

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

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

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

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

**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. 5. Well-formedness of types and environments. The rules for $\lambda_F$ exclude the grey boxes.

with kind $k$ so long as $\alpha{:}k$ is bound in the environment. The rule WF-Refn stipulates that a refined base type $b\{x : p\}$ is well-formed with base kind in some environment if the unrefined base type $b$ has base kind in the same environment and if the refinement predicate $p$ has type `Bool` in the environment augmented by binding a fresh variable to type $b$. Note that if $b \equiv \alpha$ then we can only form the antecedent $\Gamma \vdash_w \alpha\{x : \texttt{true}\} : B$ when $\alpha{:}B \in \Gamma$ (rule WF-Var), which prevents us from refining star-kinded type variables. To break a circularity in which well-formedness judgments appear in the antecedents of typing judgments and a typing judgment appears in the antecedents of WF-Refn, we use the $\lambda_F$ judgment to check that $p$ has type `Bool`. Finally, rule WF-Kind simply states that if a type $t$ is well-formed with base kind in some environment, then it is also well-formed with star kind. This rule is required by our metatheory to convert base to star kinds in type variables.

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 $\Gamma$. We write $\Gamma \vdash_F e : \tau$ to indicate that term $e$ has the (unrefined) $\lambda_F$ type $\tau$ in the (unrefined) context $\Gamma$. Fig. 6 summarizes the rules that establish typing for both $\lambda_F$ and $\lambda_{RF}$, with grey for the $\lambda_{RF}$ 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.

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

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 $\lambda_F$ we erase the refinements using $\lfloor \texttt{ty}(c) \rfloor$. The rest of the definition is similar.

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

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

$$\frac{\begin{array}{c} \Gamma \vdash e_x : t_x \\ \Gamma \vdash e : x{:}t_x \to t \end{array}}{\Gamma \vdash e\ e_x : \exists\, x{:}t_x.\ t}\textsc{T-App} \qquad \frac{\begin{array}{c} \Gamma \vdash_w t_x : k_x \\ \forall y \notin \Gamma. y{:}t_x, \Gamma \vdash e[y/x] : t\ [y/x] \end{array}}{\Gamma \vdash \lambda x.e : x{:}t_x \to t}\textsc{T-Abs} \qquad \frac{\begin{array}{c} \Gamma \vdash_w t : k \\ \Gamma \vdash e : \forall\, \alpha{:}k.\, s \end{array}}{\Gamma \vdash e[t] : s[t/\alpha]}\textsc{T-TApp}$$

$$\frac{\begin{array}{c} \forall \alpha' \notin \Gamma. \\ \alpha'{:}k, \Gamma \vdash e[\alpha'/\alpha] : t[\alpha'/\alpha] \end{array}}{\Gamma \vdash \Lambda\alpha{:}k.e : \forall\, \alpha{:}k.\, t}\textsc{T-TAbs} \qquad \frac{\begin{array}{c} \Gamma \vdash e_x : t_x \qquad \Gamma \vdash_w t : k \\ \forall y \notin \Gamma. y{:}t_x, \Gamma \vdash e[y/x] : t\,[y/x] \end{array}}{\Gamma \vdash \mathtt{let}\ x = e_x\ \mathtt{in}\ e : t}\textsc{T-Let}$$

Fig. 6. Typing rules. The judgment $\Gamma \vdash_F e : \tau$ is defined by excluding the grey boxes.

Our choice to make the typing and reduction of constants external to our language, *i.e.* given by the functions $\mathsf{ty}(c)$ and $\delta(c)$, makes our system easily extensible with further constants, including a `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* $\mathsf{ty}(c) = b\{x : p\}$*, then* $\varnothing \vdash_w \mathsf{ty}(c) : B$ *and* $\varnothing \vdash \mathtt{true} \Rightarrow p[c/x]$.
*(2) If* $\mathsf{ty}(c) = x{:}t_x \to t$ *or* $\mathsf{ty}(c) = \forall\, \alpha{:}k.\, t$*, then* $\varnothing \vdash_w \mathsf{ty}(c) : \star$.
*(3) If* $\mathsf{ty}(c) = x{:}t_x \to t$*, then for all* $v_x$ *such that* $\varnothing \vdash v_x : t_x$, $\varnothing \vdash \delta(c, v_x) : t[v_x/x]$.
*(4) If* $\mathsf{ty}(c) = \forall\, \alpha{:}k.\, t$*, then for all* $t_\alpha$ *such that* $\varnothing \vdash_w t_\alpha : k$, $\varnothing \vdash \delta_T(c, t_\alpha) : t[t_\alpha/\alpha]$.

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 $\Gamma$ 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 $\alpha$ 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.

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

***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 [2009b]. As overviewed in § 2.2, 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 $e[t]$. For the $\lambda_F$ 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 $\lambda_F$ version of the rule erases the refinements and uses $\lfloor t \rfloor$. Finally, rule T-Sub tells us that we can exchange a subtype $s$ for a supertype $t$ in a judgment $\Gamma \vdash s : t$ 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 Fig. 7, stipulates that the type $s$ is a subtype of type the $t$ in the environment $\Gamma$ and is used in the subsumption typing rule T-Sub (of Fig. 6).

***Subtyping Rules*** Rules S-Bind and S-Wit establish subtyping for existential types [Knowles and Flanagan 2009b], *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 $\Gamma$, 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 $b\{x_1 : p_1\}$ is a subtype of another $b\{x_2 : p_2\}$ in context $\Gamma$ when $p_1$ *implies* $p_2$ in the environment $\Gamma$ augmented by binding a fresh variable to the unrefined type $b$. Next, we describe how we formalized implication.

## 4.4 Implication

The *implication* judgment $\Gamma \vdash p_1 \Rightarrow p_2$ states that the implication $p_1 \Rightarrow p_2$ is (logically) valid under the assumptions captured by the context $\Gamma$. In refinement type implementations [Swamy et al. 2016; Vazou et al. 2014a], this relation is implemented as an external automated (usually SMT) solver. In non-mechanized refinement type formalizations, there have been two approaches to formalize predicate implication. Either directly reduce it into a logical implication (*e.g.* in Gordon and Fournet [2010]) or define it using operational semantics (*e.g.* in Vazou et al. [2018]). It turns out that none of these approaches can be directly encoded in a mechanized proof. The former approach is insufficient because it requires a formal connection between the (deeply embedded) terms of $\lambda_{RF}$ and the terms of the logic, which has not yet been clearly established. The second approach is more direct, since it gives meaning to implication using directly the terms of $\lambda_{RF}$, via denotational semantics. Sadly, the definition of denotational semantics for our polymorphic calculus is not currently possible: encoding type denotations as an inductive data type (or proposition in our LiquidHaskell encoding § 8) requires a negative occurrence which is not currently admitted.

***Axiomatization of Implication*** In our mechanization, following Lehmann and Tanter [2016], we encode implication as an axiomatized judgment that satisfies the requirements below.

Requirement 2. *The implication relation satisfies the following 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* $\Gamma \vdash p_1 \Rightarrow p_3$.

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

$$\frac{\Gamma \vdash t_{x2} \preceq t_{x1} \quad \forall y \notin \Gamma. y{:}t_{x2}, \Gamma \vdash t_1[y/x] \preceq t_2[y/x]}{\Gamma \vdash x{:}t_{x1} \to t_1 \preceq x{:}t_{x2} \to t_2}\textsc{S-Fun} \qquad \frac{\Gamma \vdash v_x : t_x \quad \Gamma \vdash t \preceq t'[v_x/x]}{\Gamma \vdash t \preceq \exists x{:}t_x.\, t'}\textsc{S-Wit} \qquad \frac{\forall y \notin \Gamma. \quad y{:}b, \Gamma \vdash p_1[y/x] \Rightarrow p_2[y/x]}{\Gamma \vdash b\{x : p_1\} \preceq b\{x : p_2\}}\textsc{S-Base}$$

$$\frac{\forall y \notin \mathsf{free}(t) \cup \Gamma. y{:}t_x, \Gamma \vdash t[y/x] \preceq t'}{\Gamma \vdash \exists x{:}t_x.\, t \preceq t'}\textsc{S-Bind} \qquad \frac{\forall \alpha' \notin \Gamma. \alpha'{:}k, \Gamma \vdash t_1[\alpha'/\alpha] \preceq t_2[\alpha'/\alpha]}{\Gamma \vdash \forall \alpha{:}k.\, t_1 \preceq \forall \alpha{:}k.\, t_2}\textsc{S-Poly}$$

Fig. 7. Subtyping Rules.

(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* $\Gamma \vdash p_1 \Rightarrow p_2 \land p_3$.
(5) *(Conjunction)* $\Gamma \vdash p_1 \land p_2 \Rightarrow p_1$ *and* $\Gamma \vdash p_1 \land p_2 \Rightarrow p_2$.
(6) *(Repetition)* $\Gamma \vdash p_1 \land p_2 \Rightarrow p_1 \land p_1 \land p_2$.
(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* $y{:}b, \Gamma \vdash q[y/x] \land p_1 \Rightarrow q[y/x] \land p_2$.

This axiomatic approach precisely explicates the requirements of the implication checker to establish the soundness of the entire refinement type system. In the future, we could verify that these properties hold for SMT solvers or even build other implication oracles that satisfy this contract.

## 5 $\lambda_F$ SOUNDNESS

Next, we present the metatheory of the underlying (unrefined) $\lambda_F$ that, even though it follows the textbook techniques of Pierce [2002], it is a convenient stepping stone *towards* the metatheory for (refined) $\lambda_{RF}$. In addition, the soundness results for $\lambda_F$ are used *for* our full metatheory, as our well-formedness judgments require the refinement predicate to have the $\lambda_F$ type `Bool` thereby avoiding the circularity of using a regular typing judgment in the antecedents of the well-formedness rules. The light grey boxes in Fig. 1 show the high level outline of the metatheory for $\lambda_F$ which provides a miniaturized model for $\lambda_{RF}$ but without the challenges of subtyping and existentials. Next, we describe the top-level type safety result, how it is decomposed into progress (Lemma 5.2) and preservation (Lemma 5.3) lemmas, and the various technical results that support the lemmas.

The main type safety theorem for $\lambda_F$ states that a well-typed term does not get stuck: *i.e.* either evaluates to a value or can step to another term (progress) of the same type (preservation). The judgment $\Gamma \vdash_F e : \tau$ is defined in Fig. 6 without the grey boxes, and for clarity we use $\tau$ for $\lambda_F$ types.

Theorem 5.1. *(Type Safety) If* $\varnothing \vdash_F e : \tau$ *and* $e \hookrightarrow^* e'$*, then* $e'$ *is a value or* $e' \hookrightarrow e''$ *for some* $e''$.

We prove type safety by induction on the length of the sequence of steps comprising $e \hookrightarrow^* e'$, using the preservation and progress lemmas.

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

Lemma 5.2. *(Progress) If* $\varnothing \vdash_F e : \tau$*, then* $e$ *is a value or* $e \hookrightarrow e'$ *for some* $e'$.

***Preservation*** The preservation lemma states that $\lambda_F$ typing is preserved by evaluation.

Lemma 5.3. *(Preservation) If* $\varnothing \vdash_F e : \tau$ *and* $e \hookrightarrow e'$, *then* $\varnothing \vdash_F e' : \tau$.

The proof is by structural induction on the derivation of the typing judgment. 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 5.4.

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

Lemma 5.4. *(Substitution) If* $\Gamma \vdash_F v_x : \tau_x$ *and* $\Gamma \vdash_w \lfloor t_\alpha \rfloor : k_\alpha$, *then*
*(1) if* $\Gamma', x{:}\tau_x, \Gamma \vdash_F e : \tau$ *and* $\vdash_w \Gamma$, *then* $\Gamma', \Gamma \vdash_F e[v_x/x] : \tau$ *and*
*(2) if* $\Gamma', \alpha{:}k_\alpha, \Gamma \vdash_F e : \tau$ *and* $\vdash_w \Gamma$, *then* $\Gamma'[\lfloor t_\alpha \rfloor/\alpha], \Gamma \vdash_F e[t_\alpha/\alpha] : \tau[\lfloor t_\alpha \rfloor/\alpha]$.

The proof goes by induction on the derivation tree. Because we encoded our typing rules using cofinite quantification 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 5.5. *(Weakening) If* $\Gamma_1,\Gamma_2 \vdash_F e{:}\tau$ *and* $x,\alpha \notin \Gamma_1,\Gamma_2$, *then* $\Gamma_1,x{:}\tau_x,\Gamma_2 \vdash_F e{:}\tau$ *and* $\Gamma_1,\alpha{:}k,\Gamma_2 \vdash_F e{:}\tau$.

## 6 $\lambda_{RF}$ SOUNDNESS

We proceed to the metatheory of $\lambda_{RF}$ by fleshing out the skeleton of light grey lemmas in Fig. 1 (which are similar to the $\lambda_F$ versions) and describing the three regions (§ 2.3) that establish the inversion, substitution, and narrowing properties. Type safety combines progress and preservation.

Theorem 6.1. *(Type Safety of* $\lambda_{RF}$*)*
*(1) (Type Safety) If* $\varnothing \vdash e : t$ *and* $e \hookrightarrow^* e'$, *then* $e'$ *is a value or* $e' \hookrightarrow e''$ *for some* $e''$.
*(2) (Progress) If* $\varnothing \vdash e : t$, *then* $e$ *is a value or* $e \hookrightarrow e'$ *for some* $e'$.
*(3) (Preservation) If* $\varnothing \vdash e : t$ *and* $e \hookrightarrow e'$, *then* $\varnothing \vdash e' : t$.

Next, let's see the three main ways in which the proof of Progress differs from $\lambda_F$.

### 6.1 Inversion of Typing Judgments

The vertical lined region of Fig. 1 accounts for the fact that, due to subtyping chains, the typing judgment in $\lambda_{RF}$ is not syntax-directed. First, we establish that subtyping is transitive

Lemma 6.2. *(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 6.4. As Aydemir et al. [2005], we use the Narrowing Lemma 6.6 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 6.3. *(Inversion of T-Abs, T-TAbs)*
*(1) If* $\Gamma \vdash (\lambda w.e) : x{:}t_x \to t$ *and* $\vdash_w \Gamma$, *then for all* $y \notin \Gamma$, $y{:}t_x, \Gamma \vdash e[y/w] : t[y/x]$.
*(2) If* $\Gamma \vdash (\Lambda \alpha_1{:}k_1.e) : \forall \alpha{:}k.\, t$ *and* $\vdash_w \Gamma$, *then for all* $\alpha' \notin \Gamma$, $\alpha'{:}k, \Gamma \vdash e[\alpha'/\alpha_1] : t[\alpha'/\alpha]$.

If $\Gamma \vdash (\lambda w.e) : x{:}t_x \to t$, then we cannot directly invert the typing judgment to get a typing for the body $e$ of $\lambda w.e$. Perhaps the last rule used was T-Sub, and inversion only tells us that there exists a type $t_1$ such that $\Gamma \vdash (\lambda w.e) : t_1$ and $\Gamma \vdash t_1 \preceq x{:}t_x \to t$. Inverting again, we may in fact find a chain of types $t_{i+1} \preceq t_i \preceq \cdots \preceq t_2 \preceq t_1$ which can be arbitrarily long. But the proof tree must be finite so eventually we find a type $w{:}s_w \to s$ such that $\Gamma \vdash (\lambda w.e) : w{:}s_w \to s$ and $\Gamma \vdash w{:}s_w \to s \preceq x{:}t_x \to t$

(by transitivity) and the last rule was T-Abs. Then inversion gives us that for any $y \notin \Gamma$ we have $y{:}s_w, \Gamma \vdash e : s[y/w]$. To get the desired typing judgment, we must use the Narrowing Lemma 6.6 to obtain $y{:}t_x, \Gamma \vdash e : s[y/w]$. Finally, we use T-Sub to derive $y{:}t_x, \Gamma \vdash e : t[y/w]$.

## 6.2 Substitution Lemma

The main result in the diagonal lined region of Fig. 1 is the Substitution Lemma. The biggest difference between the $\lambda_F$ and $\lambda_{RF}$ metatheories is the introduction of a mutual dependency between the lemmas for typing and subtyping. Due to this dependency, both the substitution the weakening lemmas must now be proven in a mutually recursive form for both typing and subtyping:

Lemma 6.4. *(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 main difficulty arises in substituting some type $t_\alpha$ for variable $\alpha$ in $\Gamma_1, \alpha{:}k, \Gamma_2 \vdash \alpha\{x_1 : p\} \preceq \alpha\{x_2 : q\}$ because $t_\alpha$ must be strengthened by the refinements $p$ and $q$ respectively. As with the $\lambda_F$ version, the proof requires the *Weakening* Lemma 6.5 but now both for typing and subtyping.

Lemma 6.5. *(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.

## 6.3 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 6.6. *(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 6.7 which says that both subtyping and typing is preserved after selfification.

Lemma 6.7. *(Exact Typing)*

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

## 7 REFINED DATA PROPOSITIONS

In § 8 we will present how soundness $\lambda_{RF}$ is proved in LiquidHaskell. Here we present *refined data propositions*, a novel feature of LiquidHaskell that made such a meta-theoretic proof possible. Intuitively, 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 $\lambda_{RF}$ to define the "hastype" 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   = HasTy    Env  Expr Type
  data IsSubTyPr = IsSubTy  Env  Type Type
```

Thus, `HasTy γ e t` and `IsSubTy 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 Fig. 6, with constructors that match the names of each rule.

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

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

***Step 3: Relating Evidence to its Propositions*** Next, we specify the relationship between the evidence and the proposition that it establishes, via a refinement-level *uninterpreted function*:

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

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

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

The definition stipulates that the type `HasTy γ e t` is inhabited by evidence (of type `HasTyEv`) that establishes the typing proposition `HasTyPr γ e t`. Similarly `IsSubTy γ s t` is inhabited by evidence (of type `IsSubTyEv`) that establishes the sub-typing 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, to *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 Fig. 6.

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

The constructors `TPrim` and `TSub` respectively encode the rules T-Prim and T-Sub rules (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 $\gamma, e, s, t$, 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 $\lambda_{RF}$. (Parker et al. [2019] developed a LiquidHaskell meta-theoretic proof but before data propositions and thus had to axiomatize a terminating evaluation relation; see § 10.) To implement this feature, 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.

## 8 LIQUIDHASKELL MECHANIZATION

We mechanized soundness of $\lambda_{RF}$ in both Coq 8.15.1 and LiquidHaskell 8.10.7.1, and have submitted these as anonymous supplementary material. In LiquidHaskell we use refined data propositions (§ 7) to specify the static (*e.g.* typing, subtyping, well-formedness) and dynamic (*i.e.* small-step transitions and their closure) semantics of $\lambda_{RF}$. 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 (§8.3). 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 (§ 8.1) and consists of proofs implemented as recursive functions that construct evidence to establish propositions by induction (§ 8.2). Note that while Haskell types are inhabited by diverging $\perp$ 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).

### 8.1 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 on the two argument expressions, excluding `x`, *i.e.* $\mathsf{fv}(e[v_x/x]) \subseteq \mathsf{fv}(e) \cup (\mathsf{fv}(v_x) \setminus \{x\})$. 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*.

## 8.2 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 (Theorem 6.1)*** relates the `HasTy` data proposition of § 7 with a `Step` data proposition that encodes Fig. 4 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 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.

## 8.3 Quantitative Results

We provide a full, mechanically checked proof of the metatheory in § 5 and § 6. The only facts that are assumed are the req. 2, *i.e.* the implication relation which we encoded as a data proposition and req. 1, *i.e.* assumptions about build-in primitives. Concretely, we assumed req. 1 for some constants of $\lambda_{RF}$ because it was obvious but too strenuous to mechanically prove without interactive aid.

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, each over 8000 lines across about 35 files. Currently, the whole LiquidHaskell proof can be checked in about 30 minutes, which makes interactive development difficult, especially compared to the Coq proof (§ 9) that is checked in about 30 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.

## 9 COQ MECHANIZATION

Our Coq mechanization is a translation from LiquidHaskell and was build to compare the two developments. All theorems from § 6 are proven in Coq. Req. 1 is proved (using Coq's interactive development) and req. 2 (*i.e.* the implication judgement) is axiomatized. To fairly compare the two developments in terms of effort and ergonomics, we did not used external Coq libraries because no such libraries exist yet for LiquidHaskell. Vazou et al. [2017] previously compared LiquidHaskell

| | LiquidHaskell Mechanization | | | | Coq Mechanization | |
|---|---|---|---|---|---|---|
| **Subject** | **Files** | **Time (m)** | **Spec** | **Proof** | **Spec** | **Proof** |
| Definitions | 6 | 1 | 1805 | 374 | 890 | 170 |
| Basic Properties | 8 | 4 | 646 | 2117 | 1094 | 2056 |
| $\lambda_F$ Soundness | 4 | 3 | 138 | 685 | 173 | 744 |
| Weakening | 4 | 1 | 379 | 467 | 88 | 366 |
| Substitution | 4 | 7 | 458 | 846 | 122 | 574 |
| Exact Typing | 2 | 4 | 70 | 230 | 33 | 180 |
| Narrowing | 1 | 1 | 88 | 166 | 44 | 147 |
| Inversion | 1 | 1 | 124 | 206 | 57 | 255 |
| Primitives | 3 | 4 | 120 | 277 | 88 | 500 |
| $\lambda_{RF}$ Soundness | 1 | 1 | 14 | 181 | 12 | 198 |
| **Total** | 35 | 29 | 3842 | 5549 | 2601 | 5190 |

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

and Coq as theorem provers, but their mechanizations were an order of magnitude smaller than ours and did not use data propositions that 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, $\lambda_{RF}$, 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 greatly facilitates reasoning over mutually defined and partial functions. Next, we expand upon the last two points with snippets from our mechanizations.

***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 soundness (Thm. 6.1) is encoded using Haskell's `Either` for disjunction and dependent pairs for existentials. (`Steps` is defined, using data propositions, as the closure of `Step`.)

```
soundness :: e₀:Expr → t:Type → e:Expr → HasTy Empty e₀ t → Steps e₀ e
          → Either {isVal e}  (eᵢ::Expr, Step e eᵢ)
soundness _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
     soundness e₁ t e (preservation e₀ t e₀_has_t e₁ e₀_step_e₁) e₁_eval_e
```

In Coq soundness is proved without any of the three fully applied calls above:

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

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

***Mutual Recursion*** LiquidHaskell makes it easy to define and work with mutually recursive data types, such as our typing and subtyping judgments, and to prove mutually inductive lemmas. 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, which means that structural induction using the normal `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 6.5, to do structural induction on `HasTy (concat g g')e t` we must introduce a universally quantified variable `g0` and strengthen our theorem statement with the additional 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 complex hypothesis and state the lemma as:

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

By contrast, LiquidHaskell allows us to state two separate mutually recursive lemma functions for (term variable) weakening: one for typing and one for subtyping judgments. Then we may call either lemma in their own proofs on any smaller instance of the typing (resp. subtyping) judgment.

***Partial Functions*** LiquidHaskell makes it easy to define partial Haskell functions and to prove totality with respect to the refined input types, usually automatically, without having to reason about impossible cases in mechanized proofs. For instance, our syntax does not contain an explicit `error` value, so we only want the function $\delta(c, v)$ to be defined where $c\ v$ can step in our semantics. This is straightforward in LiquidHaskell: we define a predicate `isCompat :: Prim →Value →Bool` and refine the input types of $\delta$ 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 `delta` function. However, this makes it harder to prove the determinism of our semantics due to the dependence on the proof object. The difficulty can be sidestepped by defining a partial version of $\delta$ with type `Prim →Expr →option Expr` and proving 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.

## 10 RELATED WORK

We discuss the most closely related work on the meta-theory of unrefined and refined type systems.

***Representing Binders*** Our development for $\lambda_F$ (§ 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 when formalizing refinements. 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 $\lambda_{RF}$ is *locally nameless representation* [Aydemir et al. 2008; Pollack 1993], where free variables are named, but bound variables are represented by deBruijn indices. Our metatheory 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 Fig. 4) the variable capturing problem reappears because we are substituting underneath a binder. Second, subtyping usually permits alpha-renaming of binders, which breaks a required invariant that each $\lambda_{RF}$ derivation tree is a valid $\lambda_F$ tree after erasure.

***Hybrid & Contract Systems*** Flanagan [2006] formalizes on paper a monomorphic lambda calculus with refinement types that differs from our $\lambda_{RF}$ in three ways. First, the denotational soundness methodology of Flanagan [2006] connects subtyping with expression evaluation. We could not follow this approach because encoding type denotations as a data proposition requires a negative occurrence (§ 4.4). Second, in [Flanagan 2006] type checking is hybrid: the developed system is undecidable and inserts runtime casts when subtyping cannot be statically decided. Third, the original system lacks polymorphism. Sekiyama et al. [2017] extended hybrid types with polymorphism, but unlike $\lambda_{RF}$, their system does not support semantic subtyping. For example, consider a divide by zero-error. The refined types for `div` and 0 could be given by div :: $\texttt{Int} \to \texttt{Int}\{n : n \neq 0\} \to \texttt{Int}$ and 0 :: $\texttt{Int}\{n : n = 0\}$. This system will compile `div 1 0` by inserting a cast on 0: $\langle \texttt{Int}\{n : n = 0\} \Rightarrow \texttt{Int}\{n : n \neq 0\}\rangle$, causing a definite runtime failure that could have easily been prevented statically. Having removed semantic subtyping, the metatheory of [Sekiyama et al. 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.

***Refinement Types in Coq*** Our soundness formalization follows the axiomatized implication relation of Lehmann and Tanter [2016] that decides subtyping (our rule S-Base) without formally connecting implication and expression evaluation. Lehmann and Tanter [2016]'s Coq formalization of a monomorphic lambda calculus with refinement types differs from $\lambda_{RF}$ in two ways. First, their axiomatized implication allows them to restrict the language of refinements. We allow refinements to be arbitrary program terms and intend, in the future, to connect our axioms to SMT solvers or other oracles. Second, $\lambda_{RF}$ 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 $\texttt{Int} \to \texttt{Pos}$ is subtype of $\texttt{Pos} \to \texttt{Int}$ 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.

***Metatheory in LiquidHaskell*** `LWeb` [Parker et al. 2019] also used LiquidHaskell to prove metatheory, the non-interference of $\lambda_{\text{LWeb}}$, 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 $\lambda_{\text{LWeb}}$'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 $\lambda_{RF}$ 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.

## REFERENCES

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie C. Weirich, Stephan A. Zdancewic, Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized metatheory for the masses: The POPLmark challenge. In *In TPHOLs, number 3603 in LNCS*. Springer, 50–65.

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

J. F. Belo, M. Greenberg, A. Igarashi, and B. C. Pierce. 2011a. Polymorphic Contracts. In *ESOP*.

João Filipe Belo, Michael Greenberg, Atsushi Igarashi, and Benjamin C. Pierce. 2011b. Polymorphic Contracts. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-642-19718-5_2

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

C. Flanagan. 2006. Hybrid Type Checking. In *POPL*.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. 1993. The Essence of Compiling with Continuations.. In *PLDI*.

C. Fournet, M. Kohlweiss, and P-Y. Strub. 2011. Modular code-based cryptographic verification. In *CCS*.

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

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

A. M. Kent, D. Kempe, and S. Tobin-Hochstadt. 2016. Occurrence typing modulo theories. In *PLDI*.

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. 2009a. 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

K. W. Knowles and C. Flanagan. 2009b. Compositional and decidable checking for dependent contract types. In *PLPV*.

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.

X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. 2004. Dynamic Typing with Dependent Types. In *IFIP TCS*.

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

Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press.

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

Didier Rémy. 2021. Type systems for programming languages. Course notes.

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

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 *POPL.*

N. Vazou, L. Lampropoulos, and J. Polakow. 2017. A Tale of Two Provers: Verifying Monoidal String Matching in Liquid Haskell and Coq. In *Haskell.*

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

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

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