Mechanizing Refinement Types

MICHAEL BORKOWSKI, UC San Diego, USA
NIKI VAZOU, IMDEA, 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 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 a full mechanization *of* our metatheory using the refinement-type based LiquidHaskell as a proof checker, showing how refinements can be used *for* mechanization.

1 INTRODUCTION

Refinements constrain types with logical predicates to specify new concepts. For example, the refinement type `Pos` $\doteq$ `Int`$\{v : 0 < v\}$ describes *positive* integers and `Nat` $\doteq$ `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], Racket [Kent et al. 2016] and Ruby [Kazerounian et al. 2017].

The success of refinement types relies on the combination of two essential features. First, *implicit* semantic subtyping uses semantic (SMT-solver based) reasoning to automatically convert the types of expressions without troubling the programmer for explicit type casts. For example, consider a positive expression e : `Pos` and a function expecting natural numbers f : `Nat` $\rightarrow$ `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 $f\ e$: `Pos` if, for example, f is the identity function $f : a \rightarrow a$.

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, such a development is difficult. As Sekiyama et al. [2017] observe, a naïve combination of type variables and subtyping leads to unsoundness because potentially contradicting refinements can be lost at type instantiation. Their suggested solution replaces semantic with syntactic subtyping, which is significantly less expressive. Other recent formalizations of refinement types either drop semantic subtyping [Hamza et al. 2019] or polymorphism [Flanagan 2006; Swamy et al. 2016].

In this paper we present λ_{RF}, a core calculus with a refinement type system that combines semantic subtyping with refined polymorphic type variables. We develop and establish the properties of λ_{RF} with three 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

---

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

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 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 λ_{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 λ_F, a classic System F [Pierce 2002a] 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 λ_{RF} (§ 6).

3. Verification Our final contribution is to fully mechanize the metatheory of λ_{RF} using the refinement type checker LiquidHaskell. Our formalization uses *data propositions*: a novel feature that allows encoding derivation trees for inductively defined judgments as refined data types, which lets us write plain Haskell functions (over refined data) to provide explicit witnesses that *prove* the various soundness theorems [Vazou et al. 2018]. Our proof is non-trivial, requiring 9,500 lines of code and 45 minutes of verification time, and shows, for the first time, that meta-theoretical formalizations are feasible via LiquidHaskell-style refinement typing (§ 7).

2 OVERVIEW

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

Types and Terms Both λ_F and λ_{RF} have the same syntax for *terms* e (Fig. 2). λ_F has the usual syntax for *types* t familiar from System F, while λ_{RF} additionally allows (λ_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) Γ. Additionally, both languages have *well-formedness* judgments $\Gamma \vdash_w t : k$ which say that a type t has the kind k in context Γ, by requiring that the free variables in t are appropriately bound in the environment Γ. (Though some presentations of λ_F [Pierce 2002b] eschew well-formedness judgments, they are helpful for a mechanized metatheory [Aydemir et al. 2008]). Crucially, λ_{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 Γ. 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 Γ. We take an axiomatized approach to capture precisely the properties need from an implication checking oracle for proving soundness.

One sentence explanation or remove "uninterpreted"?

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. Vertices colored light grey represent lemmas in the metatheories for λ_F and λ_{RF}. Vertices colored dark grey denote lemmas that only appear in the metatheory for λ_{RF}. An arrow shows a dependency:

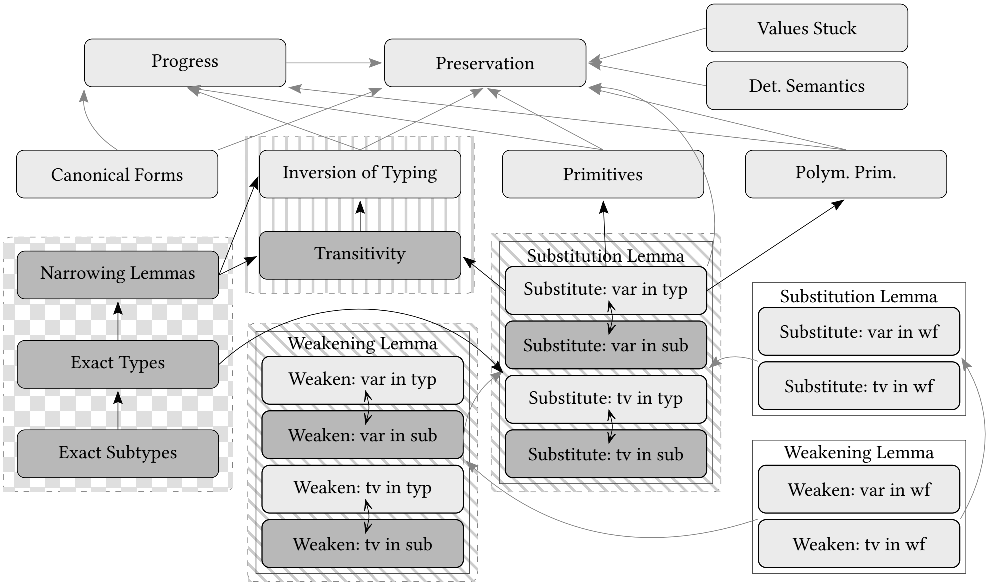

Fig. 1. Logical dependencies in the metatheory. We write "var" to abbreviate a term variable and "tv" to abbreviate a type variable.

the lemma at the *tail* is used in the proof of the lemma at the *head*. A double-headed arrow indicates a mutual dependency, *i.e.* mutually recursive proofs. Darker arrows are dependencies in λ_{RF} only.

Soundness via Preservation and Progress For both λ_{RF} and λ_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 λ_F and then outline the essential new ingredients that demonstrate soundness for the refined λ_{RF}.

2.1 Metatheory for λ_F

Progress in λ_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 λ_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

$$\Gamma \doteq f : \mathtt{Nat} \to \mathtt{Int}, e : \mathtt{Pos}$$

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 $\forall v.\ 0 < v \Rightarrow 0 \leq v$.

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

Importantly, most refinement type systems use type-constructor directed rules to destruct subtyping obligations into basic (semantic) implications. For example, in Fig. 8 the rule S-Func 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 $e : \mathtt{Nat} \to \mathtt{Pos}$ is a suitable argument for the higher order function $f : (\mathtt{Pos} \to \mathtt{Nat}) \to \mathtt{Int}$.

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.

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

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

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

The application $f\ e$ is typed as $\mathtt{Int}\{v : v = e\}$ 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 these problems are addressed. The type of $f\ e$ becomes $\exists x{:}\mathtt{Pos}.\, \mathtt{Int}\{v : v = x\}$ 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 $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$, 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}.

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.

$$
\begin{array}{rrlcl}
 & \text{Definition} & \texttt{max} & = & \lambda x\, y.\texttt{if}\ x < y\ \texttt{then}\ y\ \texttt{else}\ x \\
\text{Attempt 1:} & \textit{Monomorphism} & \texttt{max} & :: & x{:}\texttt{Int} \rightarrow y{:}\texttt{Int} \rightarrow \texttt{Int}\{v : x \leq v \wedge y \leq v\} \\
\text{Attempt 2:} & \textit{Unrefined Polymorphism} & \texttt{max} & :: & x{:}\alpha \rightarrow y{:}\alpha \rightarrow \alpha \\
\text{Attempt 3:} & \textit{Refined Polymorphism} & \texttt{max} & :: & x{:}\alpha \rightarrow y{:}\alpha \rightarrow \alpha\{v : x \leq v \wedge y \leq v\} \\
\lambda_{RF}: & \textit{Kinded Polymorphism} & \texttt{max} & :: & \forall\, \alpha{:}B.\, x{:}\alpha \rightarrow y{:}\alpha \rightarrow \alpha\{v : x \leq v \wedge y \leq v\}
\end{array}
$$

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. 2011], all the types that substitute variables should be refined. For example, if a type variable is refined with `false` (that is, $\alpha\{v : \texttt{false}\}$) 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 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 give `max` a polymorphic and precise type that naturally rejects non comparable (*e.g.* function) arguments.

2.3 From λ_F to λ_{RF}

The metatheory for λ_{RF} requires us to enrich that of λ_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 λ_{RF} is not syntax directed, and so we cannot directly invert the typing derivations of terms to get derivations for their sub-terms. For example, we cannot directly invert a derivation $\Gamma \vdash \lambda x.e : x{:}t_x \rightarrow t$ 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 address 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 let us 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 types. We cannot proceed by induction on the structure of the two subtyping judgments ($\Gamma \vdash t_1 \preceq t_2$ and $\Gamma \vdash t_2 \preceq t_3$), because we do not apply the inductive hypothesis directly to subderivations. We must first apply the substitution and narrowing lemmas in various cases, which may increase the size of the derivations used in the recursive calls (§ 6.1). Thus we must instead identify a different termination metric to show that the induction is well-founded.

Primitives	c	$::=$	`true` | `false`	*booleans*
		|	$0, 1, 2, \dots$	*integers*
		|	$\wedge, \vee, \neg, \leftrightarrow$	*boolean ops.*
		|	$\leq, =$	*polymorphic comparisons*
Values	v	$::=$	c	*primitives*
		|	$\mathrm{x}, \mathrm{y}, \dots$	*variables*
		|	$\lambda x.e$	*abstractions*
		|	$\Lambda\alpha{:}k.\mathrm{e}$	*type abstractions*
Terms	e	$::=$	v	*values*
		|	$e_1\ e_2$	*applications*
		|	$e[t]$	*type applications*
		|	`let` $x = e_1$ `in` e_2	*let-binders*
		|	$e : t$	*annotations*

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

Subtyping The biggest difference between the two metatheories is that λ_{RF} has a notion of subtyping which is crucial to making refinements practical. Subtyping complicates λ_{RF} by introducing a mutual dependency between the lemmas for typing and subtyping judgments. Recall that typing depends on subtyping due to the usual subsumption rule (T-Sub in Fig. 7) that lets us weaken the type of a term with a super-type. Conversely, subtyping depends upon typing because of the rule (S-Witn in Fig. 8) which establishes subtyping between *existential* types. Thanks to this mutual dependency, all of the lemmas from λ_F that relate to typing judgments, *i.e.* the weakening and substitution lemmas, are now mutually recursive with new versions for subtyping judgments shown in the diagonal lined region in Fig. 1.

Narrowing Finally, due to subtyping, the proofs of the typing inversion and substitution lemmas for λ_{RF} require *narrowing* lemmas that allow us to replace a type that appears 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 of λ_{RF}, we must prove narrowing for both typing and subtyping, which in turn depend on narrowing for well-formedness judgments. A few important cases of the narrowing proofs require other technical lemmas shown in the checkerboard region of Fig. 1. For example, the proof of narrowing for the "occurrence-typing" rule T-Var that crucially enables path-sensitive reasoning, uses a lemma on *selfifying* [Ou et al. 2004] the types involved in the judgments.

3 LANGUAGE

For brevity, clarity and also to cut a circularity in the metatheory (in rule WF-Refn in § 4.1), we formalize refinements using two calculi. The first is the *base* language λ_F: a classic System F [Pierce 2002a] with call-by-value semantics extended with primitive `Int` and `Bool` types and operations. The second calculus 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 and typing judgments be mutually dependent in our full language. We use the grey highlights to indicate the extensions to the syntax and rules of λ_F needed to support refinements in λ_{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. Terms are stratified into primitive *constants* and *values*. The primitives c include `Int` and `Bool` constants, primitive boolean operations, and polymorphic comparison and equality primitive. Values v are those expressions which cannot be evaluated any further, including primitive constants, binders and λ- 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 λ_F required by λ_{RF}. In λ_{RF}, only base types `Bool` and `Int` 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 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 Fig. 3 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 be distinct. Our mechanization enforces this invariant via the locally nameless representation [Aydemir et al. 2005]: free and bound variables become 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 the need for technical manipulations such as index shifting by using names instead of indices for the free variables (we discuss alternatives representations in § 8). 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 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-AppTAbs in Fig. 4. Fig. 5 summarizes how type substitution is defined, which 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 type t which is potentially refined. To do this substitution, we combine p with the type t by using $\texttt{strengthen}(t, p, x)$ which essentially conjoins the refinement p to the top-level refinement of a base-kinded t. For existential types, `strengthen` *pushes* the refinement through the existential quantifier. Function and quantified types are left unchanged as they cannot be used to instantiate a *refined* type variable (which must be of base kind).

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

Fig. 3. Syntax of Types. The grey boxes are the extensions to λ_F needed by λ_{RF}. We use τ for λ_F-only types.

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

$$
\frac{}{c\ v \hookrightarrow \delta(c, v)}\ \textsc{E-Prim}
\qquad
\frac{}{c[t] \hookrightarrow \delta_T(c, \lfloor t \rfloor)}\ \textsc{E-PrimT}
$$

$$
\frac{e \hookrightarrow e'}{e\ e_1 \hookrightarrow e'\ e_1}\ \textsc{E-App1}
\qquad
\frac{e \hookrightarrow e'}{v\ e \hookrightarrow v\ e'}\ \textsc{E-App2}
\qquad
\frac{}{(\lambda x.e)\ v \hookrightarrow e[v/x]}\ \textsc{E-AppAbs}
$$

$$
\frac{e \hookrightarrow e'}{e[t] \hookrightarrow e'[t]}\ \textsc{E-AppT}
\qquad
\frac{}{(\Lambda \alpha{:}k.e)[t] \hookrightarrow e[t/\alpha]}\ \textsc{E-AppTAbs}
$$

$$
\frac{e_x \hookrightarrow e_x'}{\texttt{let}\ x = e_x\ \texttt{in}\ e \hookrightarrow \texttt{let}\ x = e_x'\ \texttt{in}\ e}\ \textsc{E-Let}
\qquad
\frac{}{\texttt{let}\ x = v\ \texttt{in}\ e \hookrightarrow e[v/x]}\ \textsc{E-LetV}
$$

$$
\frac{e \hookrightarrow e'}{e : t \hookrightarrow e' : t}\ \textsc{E-Ann}
\qquad
\frac{}{v : t \hookrightarrow v}\ \textsc{E-AnnV}
$$

Fig. 4. The small-step semantics.

Primitives The function $\delta(c, v)$ specifies what an application $c\ \ v$ of a built-in monomorphic primitive evaluates to. The reductions are defined in a curried manner, *i.e.* we have that $\leq\ m\ n \hookrightarrow^* \delta(\delta(\leq, m), n)$. Currying gives us unary relations like $m\leq$ which is a partially evaluated version of the $\leq$ relation. We also denote by $\delta_T(=, \lfloor t \rfloor)$ and $\delta_T(\leq, \lfloor t \rfloor)$ a function specifying the reduction

$$\begin{array}{rcl}
\beta\{x:p\}[t_\alpha/\alpha] & \doteq & \beta\{x:p[t_\alpha/\alpha]\}, \alpha \neq \beta \\
(x{:}t_x \rightarrow t)[t_\alpha/\alpha] & \doteq & x{:}(t_x[t_\alpha/\alpha]) \rightarrow t[t_\alpha/\alpha] \\
(\exists\, x{:}t_x.\, t)[t_\alpha/\alpha] & \doteq & \exists\, x{:}(t_x[t_\alpha/\alpha]).\, t[t_\alpha/\alpha] \\
(\forall\, \beta{:}k.\, t)[t_\alpha/\alpha] & \doteq & \forall\, \beta{:}k.\, t[t_\alpha/\alpha] \\
\alpha\{x:p\}[t_\alpha/\alpha] & \doteq & \mathsf{strengthen}(t_\alpha, p[t_\alpha/\alpha], x)
\end{array}$$

$$\begin{array}{rcl}
\mathsf{strengthen}(\alpha\{z:q\}, p, x) & \doteq & \alpha\{z:p[z/x] \wedge q\} \\
\mathsf{strengthen}(\exists\, z{:}t_z.\, t, p, x) & \doteq & \exists\, z{:}t_z.\, \mathsf{strengthen}(t, p, x) \\
\mathsf{strengthen}(x{:}t_x \rightarrow t, \_, \_) & \doteq & x{:}t_x \rightarrow t \\
\mathsf{strengthen}(\forall\, \alpha{:}k.\, t, \_, \_) & \doteq & \forall\, \alpha{:}k.\, t
\end{array}$$

Fig. 5. Type substitution.

rules for type applications for the polymorphic built-in primitives = and ≤.

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

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

LEMMA 3.1 (DETERMINISM). *For every expression e,*

- *there exists at most one term e' such that $e \hookrightarrow e'$,*
- *there exists at most one value v such that $e \hookrightarrow^* v$, and*
- *if e is a value there is no term e' such that $e \hookrightarrow e'$.*

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 λ_F and λ_{RF}); *typing* judgments that stipulate that a term has a particular type in a given context (in λ_F and λ_{RF}); *subtyping* judgments that establish when one type can be viewed as a subtype of another (in λ_{RF}); and *implication* judgments that establish when one predicate implies another (in λ_{RF}). Next, we present the static semantics of λ_{RF} by describing each of these judgments and the rules used to establish them. We use grey to highlight the antecedents and rules specific to λ_{RF}.

Cofinite Quantification We define our rules using the cofinite quantification technique of Aydemir et al. [2008]. This technique enforces a small (but critical) restriction in the way fresh names are introduced in the antecedents of rules. For example, below we present the standard (on the left) and our (on the right) rules for type abstraction.

$$\frac{\alpha' \notin \Gamma \quad \alpha'{:}k, \Gamma \vdash e[\alpha'/\alpha] : t[\alpha'/\alpha]}{\Gamma \vdash \Lambda\alpha{:}k.e : \forall\, \alpha{:}k.\, t}\ \textsc{T-Abs-Ex} \qquad \frac{\forall \alpha' \notin \Gamma.\ \ \alpha'{:}k, \Gamma \vdash e[\alpha'/\alpha] : t[\alpha'/\alpha]}{\Gamma \vdash \Lambda\alpha{:}k.e : \forall\, \alpha{:}k.\, t}\ \textsc{T-TAbs}$$

The standard rule T-ABS-EX requires the existence of a fresh type variable name α'. Instead our co-finite quantification rule states that the rule holds for any name excluding a finite set of names (here the ones that already appear in Γ). As observed by Aydemir et al. [2008] this rephrasing simplifies the mechanization of metatheory by eliminating the need for renaming lemmas.

4.1 Well-formedness

Judgments The ternary 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 variables are only bound 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 and expression variables of the unrefined environment Γ are bound earlier in the environment. While well-formedness is not strictly required for λ_F, we found it helpful to simplify the mechanization [Rémy 2021].

Rules Fig. 6 summarizes the rules that establish the well-formedness of types and environments, with the grey highlighting the parts relevant for refinements. Rule WF-Base states that the two closed base types (`Int` and `Bool`) are well-formed and have base kind on their own or with trivial refinement `true`. Similarly, rule WF-Var says that an unrefined or trivially refined type variable α is well-formed having 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 our judgments, in which well-formedness judgments can appear in the antecedent position of typing judgments and a typing judgment would appear in the antecedent position of WF-Refn, we stipulate only a λ_F judgment for p having underlying type `Bool`. Our rule WF-Func states that a function type $x{:}t_x \rightarrow t$ is well-formed with star kind in some environment Γ if both type t_x is well-formed (with any kind) in the same environment and type t is well-formed (with any kind) in the environment Γ augmented by binding a fresh variable to t_x. Rule WF-Exis states that an existential type $\exists\, x{:}t_x.\, t$ is well-formed with some kind k in some environment Γ if both type t_x is well-formed (with any kind) in the same environment and type t is well-formed with kind k in the environment Γ augmented by binding a fresh variable to t_x. Rule WF-Poly establishes that a polymorphic type $\forall\, \alpha{:}k.\, t$ has star kind in environment Γ if the inner type t is well-formed (with any kind) in environment Γ augmented by binding a fresh type variable α to kind k. Finally, rule WF-Kind simply states that if a type t is well-formed with base kind in some environment, then it is also well-formed with star kind. This rule is required by our metatheory to convert base to star kinds in type variables.

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

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

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

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

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

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

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

$$\frac{\forall \alpha' \notin \Gamma.\quad \alpha' : k, \Gamma \vdash_w t[\alpha'/\alpha] : k_t}{\Gamma \vdash_w \forall\, \alpha{:}k.\, t : \star}\ \textsc{WF-Poly} \qquad \frac{\Gamma \vdash_w t : B}{\Gamma \vdash_w t : \star}\ \textsc{WF-Kind}$$

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

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

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

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

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

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 $0\!\leq$, $1\!\leq$, $2\!\leq$, ... For λ_F we erase the refinements using $\lfloor \text{ty}(c) \rfloor$. The rest of the definition is similar.

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

Requirement 1. *(Primitives) For every primitive c,*

(1) If $\text{ty}(c) = b\{x : p\}$, *then* $\varnothing \vdash_w \text{ty}(c) : B$ *and* $\varnothing \vdash \texttt{true} \Rightarrow p[c/x]$.

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

$$\frac{\mathsf{ty}(c) = t}{\Gamma \vdash c : t}\ \text{T-Prim} \qquad \frac{x{:}t \in \Gamma \qquad \colorbox{lightgray}{$\Gamma \vdash_w t : k$}}{\Gamma \vdash x : \colorbox{lightgray}{$\mathsf{self}(\colorbox{white}{t}, x, k)$}}\ \text{T-Var} \qquad \frac{\Gamma \vdash e : \colorbox{lightgray}{$x{:}$}t_x \to t \qquad \Gamma \vdash e_x : t_x}{\Gamma \vdash e\ e_x : \colorbox{lightgray}{$\exists x{:}t_x.\ \colorbox{white}{t}$}}\ \text{T-App}$$

$$\frac{\forall y \notin \Gamma.\quad y{:}t_x, \Gamma \vdash e[y/x] : \colorbox{lightgray}{$\colorbox{white}{$t$}\,[y/x]$} \qquad \Gamma \vdash_w t_x : k_x}{\Gamma \vdash \lambda x.e : \colorbox{lightgray}{$x{:}$}t_x \to t}\ \text{T-Abs} \qquad \frac{\Gamma \vdash e : \forall \alpha{:}k.\, s \qquad \Gamma \vdash_w t : k}{\Gamma \vdash e[t] : s[t/\alpha]}\ \text{T-TApp}$$

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

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

$$\frac{\Gamma \vdash e : t \qquad \colorbox{lightgray}{$\Gamma \vdash_w t : k$}}{\Gamma \vdash e : t : t}\ \text{T-Ann} \qquad \colorbox{lightgray}{$\dfrac{\Gamma \vdash e : s \qquad \Gamma \vdash s \preceq t \qquad \Gamma \vdash_w t : k}{\Gamma \vdash e : t}$}\ \text{T-Sub}$$

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

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

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

Typing Variables with Selfification Rule T-Var establishes that any variable x that appears as $x{:}t$ in environment Γ can be given the *selfified* type [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 judgment would result from applying rule T-Abs if T-Var merely stated that $\Gamma \vdash x : t$ whenever $x{:}t \in \Gamma$. Thus we need to 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, we introduce a "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; using the self function defined below.

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

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 with $\Gamma \vdash e\ e_x : t[e_x/x]$ in the consequent position because our version prevents the substitution of arbitrary expressions (*e.g.* functions and type abstractions) into refinements. As an alternative, we could have used A-Normal Form [Flanagan et al. 1993], but since this form is not preserved under the small step operational semantics, it would greatly compicate our metatheory, by forcing the definition of closing subtitutions for non-value expressions.

Other Typing Rules Rule T-Abs says that we can type a lambda abstraction $\lambda x.e$ at a function type $x{:}t_x \to t$ whenever t_x is well-formed and the body e can be typed at t in the environment augmented by binding a fresh variable to t_x. Our rule T-TApp states that whenever a term e has polymorphic type $\forall\,\alpha{:}k.\ s$, then for any well-formed type t with kind k in the same environment, we can give the type $s[t/\alpha]$ to the type application $e[t]$. For the λ_F variant of T-TApp, we erase the refinements (via $\lfloor t \rfloor$) before checking well-formedness and performing the substitution. The rule T-TAbs establishes that a type-abstraction $\Lambda\alpha{:}k.e$ can be given a polymorphic type $\forall\,\alpha{:}k.\ t$ in some environment Γ whenever e can be given the well-formed type t in the environment Γ augmented by binding a fresh type variable to kind k. Next, rule T-Let states that an expression `let` $x = e_x$ `in` e has type t in some environment whenever t is well-formed, e_x can be given some type t_x, and the body e can be given type t in the augmented environment formed by binding a fresh variable to t_x. Rule T-Ann establishes that an explicit annotation $e : t$ can indeed be given the type t when the underlying expression has type t and t is well-formed in the same context. The λ_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 that t is well-formed in the same context and $\Gamma \vdash s \preceq t$, which we present next.

4.3 Subtyping

Judgments Fig. 8 summarizes the rules that establish the subtyping judgment. The *subtyping* judgment $\Gamma \vdash s \preceq t$ stipulates that the type s is a subtype of type the t in the environment Γ and is used in the subsumption typing rule T-Sub (of Fig. 7).

Subtyping Rules The rule S-Func states that one function type $x_1{:}t_{x1} \to t_1$ is a subtype of another function type $x_2{:}t_{x2} \to t_2$ in a given environment Γ when both t_{x2} is a subtype of t_{x1} and t_1 is a subtype of t_2 when we augment Γ by binding a fresh variable to type t_{x2}. As usual, note that function subtyping is contravariant in the input type and covariant in the outputs. Rules S-Bind and S-Witn establish subtyping for existential types [Knowles and Flanagan 2009b], respectively 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-Witn states that if type t is a subtype of $t'[v_x/x]$ for some value v_x of type t_x, then we can discard the specific *witness* for x and quantify existentially to obtain that t is a subtype of $\exists\,x{:}t_x.\ t'$. Rule S-Poly states when one polymorphic type $\forall\,\alpha{:}k.\ t_1$ is a subtype of another polymorphic type $\forall\,\alpha{:}k.\ t_2$ in some environment Γ. The requirement is that t_1 be a subtype of t_2 in the environment where we augment Γ by binding a fresh type variable to kind k.

Refinements enter the scene in the rule S-Base which uses implication to specify that a refined base type $b\{x_1 : p_1\}$ is a subtype of another $b\{x_2 : p_2\}$ in context Γ when p_1 *implies* p_2 in the environment Γ augmented by binding a fresh variable to the unrefined type b. Next, we describe how implication is formalized in our system.

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 Γ. 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 λ_{RF} and the terms of the logic, which has not yet been clearly established. The second approach is more direct, since gives meaning to implication using directly the terms of λ_{RF}, via denotational semantics. Sadly, the definition of denotational semantics for our polymorphic calculus is not currently possible: encoding type denotations are an inductive data type (or proposition in our LiquidHaskell encoding § 7) requires a negative occurrence which is not currently admitted.

Abstracting over SMT-based Implication To bypass these problems we follow the approach of Lehmann and Tanter [2016] and encode implication as an axiomatized judgment that satisfies the below requirements.

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* $\Gamma \vdash p_1 \Rightarrow p_3$.
3. *(Introduction) If* $\Gamma \vdash p_1 \Rightarrow p_2$ *and* $\Gamma \vdash p_1 \Rightarrow p_3$, *then* $\Gamma \vdash p_1 \Rightarrow p_2 \wedge p_3$.
4. *(Conjunction 1)* $\Gamma \vdash p_1 \wedge p_2 \Rightarrow p_1$.
5. *(Conjunction 2)* $\Gamma \vdash p_1 \wedge p_2 \Rightarrow p_2$.
6. *(Repetition)* $\Gamma \vdash p_1 \wedge p_2 \Rightarrow p_1 \wedge p_1 \wedge p_2$.
7. *(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* $\Gamma_1, x\!:\!s_x, \Gamma_2 \vdash p_1 \Rightarrow p_2$.
8. *(Weakening) 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* $\Gamma_1, a\!:\!k, \Gamma_2 \vdash p_1 \Rightarrow p_2$.
9. *(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* $\Gamma_1[v_x/x], \Gamma_2 \vdash p_1[v_x/x] \Rightarrow p_2[v_x/x]$.
10. *(Subst II) If* $\Gamma_1, a\!:\!k, \Gamma_2 \vdash p_1 \Rightarrow p_2$ *and* $\Gamma_2 \vdash_w t : k$, *then* $\Gamma_1[t/a], \Gamma_2 \vdash p_1[t/a] \Rightarrow p_2[t/a]$.
11. *(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 axiomatic approach precisely explicates the properties that are required of of the implication checker in order to establish the soundness of the entire refinement type system. In the future, we can look into either verifying that these properties hold for SMT-based checkers, or even build other kinds of implication oracles that adhere to this contract.

5 λ_F SOUNDNESS

Next, we present the metatheory of the underlying (unrefined) λ_F that, even though it follows the textbook techniques of Pierce [2002a], it is a convenient stepping stone *towards* the metatheory for (refined) λ_{RF}. In addition, the soundness results for λ_F are used *for* our full metatheory, as our well-formedness judgments require the refinement predicate to have the λ_F type `Bool` thereby avoiding the circularity of using a regular typing judgment in the antecedents of the well-formedness rules. The light grey boxes in Fig. 1 show the high level outline of the metatheory for λ_F which provides a miniaturized model for λ_{RF} but without the challenges of subtyping and existentials. Next, we describe the top-level type safety result, how it is decomposed into progress (§ 5.1) and preservation (§ 5.2) lemmas, and the various technical results that support the lemmas.

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

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

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

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

Fig. 8. Subtyping Rules.

The main type safety theorem for λ_F states that a well-typed term does 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. 7 without the grey boxes, and for clarity we use τ for λ_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.

5.1 Progress

The progress lemma says that a well-typed term is either 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'.*

Proof of progress requires a *Canonical Forms* lemma (Lemma 5.3) which describes the shape of well-typed values and some key properties about the built-in *Primitives* (Lemma 5.5). We also implicitly use an *Inversion of Typing* lemma (Lemma 5.4) which describes the shape of the type of well-typed terms and its subterms. For λ_F, unlike λ_{RF}, this lemma is trivial because the typing relation is syntax-directed.

Lemma 5.3. *(Canonical Forms)*

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

Lemma 5.4. *(Inversion of Typing)*

(1) If $\Gamma \vdash_F c : \tau$, then $\tau = \lfloor \mathsf{ty}(c) \rfloor$.
(2) If $\Gamma \vdash_F x : \tau$, then $x{:}\tau \in \Gamma$.
(3) If $\Gamma \vdash_F e\ e_x : \tau$, then there is some type τ_x such that $\Gamma \vdash_F e : \tau_x \to \tau$ and $\Gamma \vdash_F e_x : \tau_x$.
(4) If $\Gamma \vdash_F \lambda x.e : \tau$, then $\tau = \tau_x \to \tau'$ and $y{:}\tau_x, \Gamma \vdash_F e[y/x] : \tau'$ for any $y \notin \Gamma$ and well-formed τ_x.
(5) If $\Gamma \vdash_F e[t] : \tau$, then there is some type σ and kind k such that $\Gamma \vdash_F e : \forall\, \alpha{:}k.\, \sigma$ and $\tau = \sigma[\lfloor t \rfloor/\alpha]$.
(6) If $\Gamma \vdash_F \Lambda\alpha{:}k.e : \tau$, then there is some type τ' and kind k such that $\tau = \forall\, \alpha{:}k.\, \tau'$ and $\alpha'{:}k, \Gamma \vdash_F e[\alpha'/\alpha] : \tau'[\alpha'/\alpha]$ for some $\alpha' \notin \Gamma$.

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

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

Lemma 5.5. *(Primitives) For each built-in primitive c,*

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

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

Lemmas 5.3 and 5.4 are proved without induction by inspection of the derivation tree, while lemma 5.5 relies on the Primitives Requirement 1.

5.2 Preservation

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

Lemma 5.6. *(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. For applications of primitives, preservation requires the Primitives Lemma 5.5, while the general case needs a Substitution Lemma 5.7.

Substitution Lemma To prove that types are preserved when a lambda or type abstraction is applied, we must show that the substituted result has the same type, which is established by the substitution lemma:

Lemma 5.7. *(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* $\Gamma', \Gamma \vdash_F e[v_x/x] : \tau$*.*

(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 (§ 4) the proof does not require a renaming lemma, but the rules that lookup environments (rules T-Var and WF-Var) do need a lemma the below *Weakening* Lemma 5.8.

Lemma 5.8. *(Weakening Environments) If* $\Gamma_1, \Gamma_2 \vdash_F e : \tau$ *and* $x, \alpha \notin \Gamma_1, \Gamma_2$*, then*

(1) $\Gamma_1, x{:}\tau_x, \Gamma_2 \vdash_F e : \tau$*.*

(2) $\Gamma_1, \alpha{:}k, \Gamma_2 \vdash_F e : \tau$*.*

6 λ_{RF} SOUNDNESS

We proceed to the metatheory of λ_{RF} by fleshing out the skeleton of light grey lemmas in Fig. 1 (which have similar statements to the λ_F versions) and describing the three regions (§ 2.3) needed to establish the properties of inversion, substitution, and narrowing.

Type Safety The top-level type safety theorem, like λ_F, combines progress and preservation

Theorem 6.1. *(Type Safety) If* $\varnothing \vdash e : t$ *and* $e \hookrightarrow^* e'$*, then* e' *is a value or* $e' \hookrightarrow e''$ *for some* e''*.*

Lemma 6.2. *(Progress) If* $\varnothing \vdash e : t$*, then* e *is a value or* $e \hookrightarrow e'$ *for some* e'*.*

Lemma 6.3. *(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 Lemma 6.2 differs from λ_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 λ_{RF} is not syntax-directed. First, we establish that subtyping is transitive

LEMMA 6.4. *(Transitivity) If* $\Gamma \vdash_w t_1 : k_1, \Gamma \vdash_w t_3 : k_3, \vdash_w \Gamma, \Gamma \vdash t_1 \preceq t_2$, *and* $\Gamma \vdash t_2 \preceq t_3$, *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-WITN and the latter is S-BIND, we require the Substitution Lemma 6.6. As Aydemir et al. [2005], we use the Narrowing Lemma 6.8 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 6.5. *(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* $y \notin \Gamma$ *we have* $y{:}t_x, \Gamma \vdash e[y/w] : t[y/x]$.
(2) If $\Gamma \vdash (\Lambda \alpha_1{:}k_1.e) : \forall \alpha{:}k.\, t$ *and* $\vdash_w \Gamma$, *then for every* $\alpha' \notin \Gamma$ *we have* $\alpha'{:}k, \Gamma \vdash e[\alpha'/\alpha_1] : t[\alpha'/\alpha]$.

If $\Gamma \vdash (\lambda w.e) : x{:}t_x \to t$, then we cannot directly invert the typing judgment to get a typing judgment for the body e of $\lambda w.e$. Perhaps the last rule used was T-SUB, and inversion only tells us that there exists a type t_1 such that $\Gamma \vdash (\lambda w.e) : t_1$ and $\Gamma \vdash t_1 \preceq x{:}t_x \to t$. Inverting again, we may in fact find a chain of types $t_{i+1} \preceq t_i \preceq \cdots \preceq t_2 \preceq t_1$ which can be arbitrarily long. But the proof tree must be finite so eventually we find a type $w{:}s_w \to s$ such that $\Gamma \vdash (\lambda w.e) : w{:}s_w \to s$ and $\Gamma \vdash w{:}s_w \to s \preceq x{:}t_x \to t$ (by transitivity) and the last rule was T-ABS. Then inversion gives us that for any $y \notin \Gamma$ we have $y{:}s_w, \Gamma \vdash e : s[y/w]$. To get the desired typing judgment, we must use the Narrowing Lemma 6.8 to obtain $y{:}t_x, \Gamma \vdash e : s[y/w]$ and 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 λ_F and λ_{RF} metatheories is the introduction of a mutual dependency between the lemmas for typing and subtyping judgments. Due to this dependency, the substitution lemma, and the weakening lemma on which it depends must now be proven in a mutually recursive form for both typing and subtyping judgments:

LEMMA 6.6. *(Substitution)*

- *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* $\Gamma_1[v_x/x], \Gamma_2 \vdash s[v_x/x] \preceq t[v_x/x]$.
- *If* $\Gamma_1, x{:}t_x, \Gamma_2 \vdash e : t, \vdash_w \Gamma_2$, *and* $\Gamma_2 \vdash v_x : t_x$, *then* $\Gamma_1[v_x/x], \Gamma_2 \vdash e[v_x/x] : t[v_x/x]$.
- *If* $\Gamma_1, \alpha{:}k, \Gamma_2 \vdash s \preceq t, \vdash_w \Gamma_2$, *and* $\Gamma_2 \vdash_w t_\alpha : k$, *then* $\Gamma_1[t_\alpha/\alpha], \Gamma_2 \vdash s[t_\alpha/\alpha] \preceq t[t_\alpha/\alpha]$.
- *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_α for variable α in $\Gamma_1, \alpha{:}k, \Gamma_2 \vdash \alpha\{x_1 : p\} \preceq \alpha\{x_2 : q\}$ because we must deal with strengthening t_α by the refinements p and q respectively. As with the λ_F metatheory, the proof of the substitution lemma does not require renaming, but does require a lemmas that let us *weaken* environments (Lemma 6.7) in typing and subtyping judgments.

LEMMA 6.7. *(Weakening Environments) 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 6.8. *(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 6.9 which says that a subtyping judgment $\Gamma \vdash s \preceq t$ is preserved after selfification on both types. Similarly whenever we can type a value v at type t then we also type v at the type t selfified by v.

Lemma 6.9. *(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* $\Gamma \vdash v : \mathrm{self}(t, v, k)$.

7 MECHANIZATION

We mechanized our results using LiquidHaskell [Vazou et al. 2014b] and our implementation is submitted as anonymous supplementary material. Our mechanization, which is partially simplified by SMT-automation (§ 7.1), introduces a novel LiquidHaskell feature that we call *data propositions*, which lets us encode rules as refined types (§ 7.2), derivation trees as refined data types (§ 7.3 and § 7.4 for cofinite quantified encoding), and proofs as inductive functions (§ 7.5). Note that while Haskell types are inhabited by diverging ⊥ values, LiquidHaskell checks that all definitions are terminating, and hence that the induction is well-founded.

7.1 SMT Solvers 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 set theory, which we used simplify the establishment of variable invariants.

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

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

In the above (incomplete) definition, `S` is used to qualify the standard `Data.Set` Haskell library. LiquidHaskell embeds the functions of `Data.Set` to SMT set operators (encoded as a map to booleans). For example, `S.union` is treated as the logical set union operator ∪. Further, we lift `fv` in the refinement logic using `{-@ measure fv @-}` annotation (and in general we use the special comments `{-@ ... @-}`) to provide LiquidHaskell specific annotations. The measure definition serves two purposes: first it defines the logical function `fv` and second it *decidably* embeds the measure definition to each `Expr` constructor. This embedding, combined with the SMT, set theory knowledge, let us prove "for free" properties about expressions' free variables.

Intrinsic Verification Below we define the function `subFV x vx e` to substitute in `e` the free variable `x` with the `vx` and give it a refinement type that 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 post condition 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*, that is the definition of `subFV` constitutes the proof (no user aid is required) and, importantly, the specification is automatically established each time the function `subFV` is called. So, the user does not have to provide explicit hints to reason about free variables of substituted expressions.

The `fv` function is just an example of SMT-based proof simplification. In the specification of `subFV`, we see another example. The Haskell boolean function `isVal` that states that an expression is a value, is also declared as a **`measure`**, and we intrinsically prove that that value property is preserved by substitution.

Freshness Our combination of Haskell's and SMT's sets let us easily define a `fresh` function, which has been challenging in many theorem provers[1]. `fresh xs` returns a variable that provably does not belong to its input `xs`.

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

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

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

The `fresh` function returns `n`: the maximum element of the set increased by one. To compute the maximum element we convert the set to a list and use the inductively defined `maxs` functions. To prove `fresh`'s intrinsic specification we use an extrinsic, *i.e.* explicit, lemma that the `n` is above the maximum element. This extrinsic lemma is trivially proved by induction and SMT automation.

Next, we present how we advanced such extrinsic proofs to encode and prove soundness of λ_{RF}.

7.2 Propositions as Refined Types

The elements of λ_{RF} are deeply embedded as Haskell data definitions. For example, we defined the data types `Expr`, `Type`, and `Env` to encode the expressions (Fig. 2), types, and environment (Fig. 3) of our calculus. To encode the judgments of λ_{RF}, we defined the type `DataProp`:

```
  data DataProp = HasType   Env   Expr Type   -- Γ ⊢ e : t (Fig. 7)
```

---

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

```
                  | IsSubtype Expr Type Type  -- Γ ⊢ t ≼ t′ (Fig. 8)
                  | Step      Expr Expr       -- e ↪ e′ (Fig. 4)
                  | EvalsTo   Expr Expr       -- e ↪* e′
                  ...
```

For example, `typeOne` below states that `1` has the integer singleton type in the empty environment.

```
typeOne :: DataProp
typeOne = HasType Empty (EPrim (PInt 1)) t  -- ∅ ⊢ 1 : Int{ν : ν = 1}
  where
    t   = TRefn TInt 0 (EApp (EApp (EPrim EEq) (EVar 0)) (EPrim (PInt 1)))
```

We define the `ProofOf` operator that turns propositions into refined types.

```
  type ProofOf (e :: DataProp) = { p:a | prop p = e }
```

and `prop` is an *uninterpreted* function in the refinement logic [Vazou et al. 2018]

```
  prop :: a → DataProp
```

The type `ProofOf e` desugars into `{p:a | prop p = e}` for some type `a` which is filled in by GHC and is irrelevant to the proposition statement. Thus, any expression of type `ProofOf e` is a *witness* that explicitly demonstrates why, *i.e. proves*, the proposition `e` holds. As `prop` is an uninterpreted function, *i.e.* has no definition, the only way to generate such proofs is via the refined data proposition constructors, as we describe next.

7.3 Derivation Trees as Refined Data Types

To construct proposition witnesses use define data types that encode λ_{RF} derivation rules. As an example, we defined the refined data type `HasType` to encode the typing rules of Fig. 7. `HasType` has one data constructor for each typing rule, for example `TPrim` and `TSub` below, respectively encode the rules T-Prim and T-Sub rules (with the latter being simplified here for exposition).

```
data HasTypeT where               data HasTypeT where
   TPrim :: Env   → Prim             TPrim :: γ:Env   → c:Prim
         → HasTypeT                            → ProofOf(HasType γ (Prim c) (ty c))
 | TSub  :: Env   → Expr           | TSub :: γ:Env    → e:Expr
         → Type  → HasTypeT                  → t:Type  → ProofOf(HasType γ e t)
         → Type  → IsSubtypeT                → t':Type → ProofOf(IsSubtype γ t t')
         → HasTypeT                          → ProofOf(HasType γ e t')
 | ...                            | ...
```

On the left, the Haskell data type defines the structure of the derivation tree, while on the right the refinements propagate propositions. As an example, below we use the primitive and subtyping constructors to construct the witness (equivalently derivation tree) that `1` is a positive integer.

```
onePos :: ProofOf (HasType Empty (EPrim (PInt 1)) {v:Int | 0 < v})
onePos = TSub Empty (EPrim (PInt 1))
              {v:Int | 1 = v} (TPrim Empty (PInt 1)) -- ∅ ⊢ 1 : Int{ν : ν = 1}
              {v:Int | 0 < v} (SBase ...) -- ∅ ⊢ Int{ν : ν = 1} ≼ Int{ν : 0 < ν}
```

7.4 Cofinite Quantification

To encode the rules that need a fresh free variable name we use the cofinite quantification of Aydemir et al. [2008], as discussed in § 4. Figure 9 presents this encoding using the T-Abs rule as an example.

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

-- Cofinitely Quantified Rule
TAbsCQ  :: γ:Env → t_x:Type → e:Expr → t:Type
        → l:[VName]
        → ( y:{VName | y ∉ S.fromList l } →
            ProofOf (HasType ((y,t_x):γ) (unbind y e) (unbindT y t)) )
        → ProofOf (HasType γ (ELam e) (TFunc t_x t))

-- Final Rule: Cofinitely Quantified and Explicit Size
TAbs    :: n:Nat → γ:Env → t_x:Type → e:Expr → t:Type
        → l:[VName]
        → ( y:{VName | y ∉ S.fromList l } →
            ProofOfN n (HasType ((y,t_x):γ) (unbind y e) (unbindT y t)) )
        → ProofOfN (n+1) (HasType γ (ELam e) (TFunc t_x t))

-- Note: All rules should include k_x:Kind → ProofOf (WfType γ t_x k_x)
-- in the first line, which we ommit for clarity.
```

Fig. 9. Encoding of Cofinitly Quantified Rules.

The standard abstraction rule (rule T-Abs-Ex in § 4) requires to provide a concrete fresh name, whish is encoded in the second line of `TAbsEx` as the `y:{VName | y ∉ dom γ }` argument.

The cofinitely equalified encoding of the rule `TAbsCQ`, instead, states that there exists a specified finite set of excluded names, namely `l`, and requires that the sub-derivation holds for any name `y` that does not belong in `l`. That is, the premise is turned into a function that, given the name `y`, returns the subderivation. This encoding greatly simplifies our mechanization, since the premises are no more tied to concrete names, eliminating the need for renaming lemmas.

But, this encoding introduces an interesting challenge in the construction of proofs by induction on the derivation tree (§ 7.5). LiquidHaskell cannot conclude that the size of the derivation subtree is independent of y, which makes it impossible accept inductive hypotheses on quantified subtrees. This challenge does not appear in the Coq formalization of Aydemir et al. [2008], since Coq can generate induction principles that works over cofinitely quantified inductive hypotheses.

I really do not understand how l connects with gamma... I think we need to say something about it

To address this challenge, in LiquidHaskell, we explicitly captured the size of the quantified rules using a ghost size argument. Concretely, we defined the `ProofOfN n e` operator to be a `ProofOf e` with size bounded by `n`, where `size` is an uninterpreted function:

```
measure size :: a → Nat
type ProofOfN (n :: Nat) (e :: DataProp) = { p: ProofOf e | size p ≤ N }
```

Our final `TAbs` rule takes the extra ghost size argument `n` and ensures that the size of the conclusion is bounded by `n+1`, if the size of the premise is bounded by `n`. With this encoding induction on

Also, why encoded as list and then

derivation trees is permitted, but at the extra cost of providing an explicit size argument to quantified judgments. We believe that the price of renaming-lemma elimination worths this cost.

7.5 Inductive Proofs as Recursive Functions

The majority of our proofs are by induction on derivations. These proofs are written as recursive Haskell functions that operate over the refined data types reifying the respective derivations. 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 6.3) is proved by induction on the derivation tree. The subtyping case requires an induction while the primitive case is impossible (Lemma 3.1):

```
preservation :: e:Expr → t:Type → ProofOf (HasType Empty e t)
             → e':Expr → ProofOf (Step e e')
             → ProofOf (HasType Empty e' t)

preservation _e _t (TSub n Empty e t' e_has_t' t t'_sub_t) e' e_step_e'
  = TSub n' Empty e' t' e'_has_t' t t'_sub_t
      where
        e'_has_t' = preservation e t' e_has_t' e' e_step_e'
        n'        = typSize e'_has_t'
preservation e _ (TPrim _ _) e' step
  = impossible "value" ? lemValStep e e' step -- e ↪ e' ⇒ ¬(isVal e)
preservation e _ (TAbs {}) e' step
  = impossible "value" ? lemValStep e e' step -- e ↪ e' ⇒ ¬(isVal e)
...

impossible :: {v:String | false} → a
lemValStep :: e:Expr → e':Expr → ProofOf (Step e e') → {¬(isVal e)}
(?)        :: a → b → b
```

In the `TSub` case we note that LiquidHaskell knows that the expression argument `_e` is equal to the the subtyping parameter `e`. Further, the termination checker will ensure that the inductive call happens on the smaller derivation subtree. The `TPrim` case goes by contradiction since primitives cannot step. We separately proved that values cannot step in the `lemValStep` lemma, which here is combined with the fact that `e` is a value to allow the call of the false-precondition `impossible`. Finally, LiquidHaskell's totality checker ensures all the cases of `HasTypeT` are covered, and the termination checker ensures the proof is well-founded.

There was no need to state or prove a size bound on `ProofOf (HasType Empty e' t)` because the preservation lemma does not use any cofinitely quantified rules.

Progress (Theorem 6.2) ensures that a well-typed expression is a value *or* there exists an expression to which it steps. To express this claim we used Haskell's `Either` to encode disjunction that contain pairs (refined to be dependent) to encode existentials.

```
progress :: e:Expr → t:Type → ProofOf (HasType Empty e t)
         → Either {isVal e}  (e'::Expr, ProofOf (Step e e'))

progress _e _t (TSub Empty e t' e_has_t' t t_sub_t')
```

```
    = progress e t' e_has_t'
  progress _e _t (TPrim _ _)
    = Left ()
  progress _e _t (TAbs _ _)
    = Left ()
  ...
```

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

Soundness (Theorem 6.1) ensures that a well-typed expression will not get stuck, that is, it will either reach a value or keep evaluating. We encode evaluation as a refined type `EvalsTo e₀ e` with a reflexive and a recursive constructor. Our soundness proof goes by induction on the length of the evaluation sequence.

```
soundness :: e₀:Expr → t:Type → ProofOf (HasType Empty e₀ t)
          → e:Expr → ProofOf (EvalsTo e₀ e)
          → Either {isVal e}  (eᵢ::Expr, ProofOf (Step e eᵢ))

soundness _e₀ t e₀_has_t _e 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 (preservation e₀ t e₀_has_t e₁ e₀_step_e₁) e 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.

7.6 Mechanization Details

We provide a full, mechanically checked proof of the metatheory in § 5 and § 6. The only facts taken for granted are the requirements on built-in primitives (1) and on the implication relation (2).

Representing Binders In our mechanization, we use the *locally-nameless representation* [Aydemir et al. 2008; Charguéraud 2012]. Free variables and bound variables are taken to be separate syntactic objects, so we do not need to worry about alpha renaming of free variables to avoid capture in substitutions. We also use de Bruijn indices only for bound variables. This enables us to avoid taking binder names into account in the strengthen function used to define substitution (Fig. 5).

Quantitative Results In Table 1 we give the empirical details of our metatheory, which was checked using LiquidHaskell version 0.8.10.7.1 and an Intel Xeon W-2133 processor with 6 physical cores and 128 GB of RAM. Our mechanized proof is substantial, spanning about 9500 lines distributed over about 34 files. Currently, the whole proof can be checked in about 44 minutes, which can make interactive development difficult. While incremental modular checking provides a modicum of interactivity, improving the ergonomics, *i.e.* verification time and providing actionable error messages, remains an important direction for future work.

8 RELATED WORK

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

Soundness of System F Our development for λ_F (§ 5) follows the standard presentation of System F's metatheory by Pierce [2002a]. The main difference between the two metatheories is that ours

Subject	**Files**	**Spec. (LOC)**	**Proof (LOC)**	**Time (mins)**
Definitions	6	1832	434	3
Basic Properties	8	653	2115	9
λ_F Soundness	3	122	525	3
Weakening	4	392	493	3
Substitution	4	501	870	6
Exact Typing	2	76	235	7
Narrowing	1	102	185	1
Inversion of Typing	1	141	211	3
Primitives	3	122	283	8
λ_{RF} Soundness	1	14	188	2
Total	34	3955	5539	44

Table 1. Empirical details of our mechanization. We partition the development into sets of modules pertaining to different region of Fig. 1, and for each region separate the lines of specification (*e.g.* definitions and lemma statements) from those needed for proofs.

includes well-formedness of types and environments, which help with mechanization [Rémy 2021] and are crucial when formalizing refinements.

Variable Representations One of the main challenges in mechanization of metatheories 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], that is, free variables are named, but the bound variables are represented by syntactically distinct deBruijn indices. We chose this representation because it clearly addresses the following two problems with named bound variables but nevertheless our metatheory still resembles the paper and pencil proofs (that we performed before mechanization): First, when different refinements are strengthened (as in Fig. 5) the variable capturing problem reappears because we are substituting underneath a binder. Second, subtyping usually permits alpha-renaming of variables, which breaks a required invariant that each λ_{RF} derivation tree is a valid λ_F tree after erasure.

Hybrid & Contract Systems Flanagan [2006] formalizes a monomorphic lambda calculus with refinement types that differs from our λ_{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 λ_{RF}, their system does not support semantic subtyping. For example, consider a divide by zero-error. The refined types for `div` and 0 could be given by div :: $\texttt{Int} \to \texttt{Int}\{n : n \neq 0\} \to \texttt{Int}$ and 0 :: $\texttt{Int}\{n : n = 0\}$. This system will compile `div 1 0` by inserting a cast on 0: $\langle \texttt{Int}\{n : n = 0\} \Rightarrow \texttt{Int}\{n : n \neq 0\}\rangle$, causing a definite runtime failure that could have easily been prevented statically. Having removed semantic subtyping, the metatheory of [Sekiyama et al. 2017] is highly simplified. Finally, neither of the two systems comes with a machine checked proof.

CHECK: removed denotational

Decidable Systems Static refinement type systems (as summarized by Jhala and Vazou [2020]) usually restrict the definition of predicates to quantifier-free first-order formulae that can be *decided* by SMT solvers. This restriction though 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. Here, we allow predicates to be any language term (including lambdas) to prove soundness via preservation and progress, but our meta-theoretical results trivially apply to systems that, for efficiency of implementation, restrict their source languages.

Refinement Types in Coq Our soundness formalization follows the axiomatized implication of Lehmann and Tanter [2016]. They axiomatize a logical implication relation that decides subtyping (our rule S-Base) which provides no formal connection between subtyping and expression evaluation. Lehmann and Tanter [2016]'s Coq formalization of a monomorphic lambda calculus with refinement types differs from λ_{RF} in two ways. First, their axiomatized implication allows them to arbitrarily 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, λ_{RF} includes polymorphism, existentials, and selfification which are crucial for path- and context-sensitive refinement typing, but make the metatheory more challenging.

System FR Hamza et al. [2019] present a polymorphic, refined language with a mechanized metatheory of comparable size (about 20,000 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 different notion of subtyping, their mechanization did not require either the indirection of denotational soundness or the use of an implication proving oracle.

9 CONCLUSIONS & FUTURE WORK

We presented and formalized soundness of λ_{RF}, a core refinement calculus that combines semantic subtyping, existential types, and parametric polymorphism, which are critical for practical refinement type systems but have never been formalized before combined. Our meta-theory is mechanized in LiquidHaskell, making use of SMT to automate various tedious invariants about variables and, for first time, using the novel feature of refined data propositions to encode inductive predicates corresponding to typing derivations in LiquidHaskell. Based on our results, we envision at least two distinct lines of work on mechanizing metatheory *of* and *with* refinement types.

1. Mechanization of Refinements First, λ_{RF} covers a crucial but small fragment of the features of modern refinement type checkers. It would be interesting to extend the language to include features like refined datatypes, and abstract and bounded refinements. 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 Second, 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 — 9000 lines across 34 files take about in 45 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.

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

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

Arthur Charguéraud. 2012. The Locally Nameless Representation. *J. Autom. Reason.* 49, 3 (2012), 363–408. https://doi.org/10.1007/s10817-011-9225-2

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. 2020. Refinement Types: A Tutorial. *CoRR* abs/2010.07763 (2020). arXiv:2010.07763 https://arxiv.org/abs/2010.07763

Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak. 2017. Refinement Types for Ruby. *CoRR* abs/1711.09281 (2017). arXiv:1711.09281 http://arxiv.org/abs/1711.09281

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 (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.

George C. Necula. 1997. Proof carrying code. In *POPL 97: Principles of Programming Languages*. ACM, 106–119.

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

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

Benjamin C. Pierce. 2002a. *Types and Programming Languages*.

B. C. Pierce. 2002b. *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)*, Henk Barendregt and Tobias Nipkow (Eds.), Vol. 806. 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

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

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*.

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 (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 (FPCA '89)*. Association for Computing Machinery, New York, NY, USA, 347–359. https://doi.org/10.1145/99370.99404