A Hoare Logic Style Refinement Types Formalisation

Zilin Chen
UNSW
Sydney, Australia
zilin.chen@student.unsw.edu.au

Abstract

Refinement types is a lightweight yet expressive tool for specifying and reasoning about programs. The connection between refinement types and Hoare logic has long been recognised but the discussion remains largely informal. In this paper, we present a Hoare triple style Agda formalisation of a refinement type system on a simply-typed λ-calculus restricted to first-order functions. In our formalisation, we interpret the object language as shallow Agda terms and use Agda's type system as the underlying logic for the type refinement. To deterministically typecheck a program with refinement types, we reduce it to the computation of the weakest precondition and define a verification condition generator which aggregates all the proof obligations that need to be fulfilled to witness the well-typedness of the program.

CCS Concepts: • **Theory of computation** → **Type theory**; **Hoare logic**; **Logic and verification**; **Denotational semantics**; **Pre- and post-conditions**; *Program verification*; *Abstraction*.

Keywords: refinement type, Hoare logic, Agda, weakest precondition

ACM Reference Format:
Zilin Chen. 2022. A Hoare Logic Style Refinement Types Formalisation. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '22), September 11, 2022, Ljubljana, Slovenia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3546196.3550162

1 Introduction

Program verification has been widely adopted by software developers in the past few decades, and is becoming standard practice when it comes to the development of safety-critical software. Refinement types[1], due to its tight integration into the programming language, has seen the growing popularity among language designers and users. It is effective yet easy to harness by programmers who are less exposed to more sophisticated forms of formal verification. Programmers can annotate their code with types that include predicates which further restrict the inhabitants of that type. For instance, $\{\nu : \mathbb{N} \mid \nu > 0\}$ is a type that only includes positive natural numbers. We typically call the type being refined, namely $\mathbb{N}$ here, the *base type*, and the logical formula the *refinement predicate*.

Refinement types introduce several complications. Typically, a refinement type system supports *dependent functions*, which are similar to those in a dependent type system [19]. A dependent function allows the refinement predicate of the return type to refer to the value of the function's arguments. Such term-dependency also results in the typing contexts being telescopic, meaning that a type in the context can refer to variables in earlier entries of that context.

Another source of complexity in refinement type systems is solving the logical entailment which determines the subtyping relation between two types. Usually, some syntactic or semantic rewriting tactics will be employed to carefully transform the entailment into a certain form, that facilitates the automatic discharge of proof obligations using an Satisfiability Modulo Theories (SMT) solver. To ensure that SMT-solving is decidable, language designers typically need to restrict the logic used for expressing refinement predicates. For instance, in Liquid Haskell [37], the quantifier-free logic of equality, uninterpreted functions and linear arithmetic (QF-EUFLIA) is chosen to ensure decidability.

In the rich literature on refinement types, while there are notable exceptions (e.g. [17]), due to the complexity of refinement type systems, the development remains largely informal[2] and rather ad hoc. For instance, the typing rules of each variant of a refinement type system can be subtly different, whereas the underlying reasons for the difference are not always systematically analysed and clearly attributed.

The connections between refinement types and Hoare logic have long been recognised. For example, the work by Jhala and Vazou [12] indicates that "refinement types can be viewed as a generalization of Floyd-Hoare style program logics." The monolithic assertions on the whole program states in Hoare logic can be decomposed into more fine-grained refinements on the values of individual terms. Pre- and post-conditions correspond directly to functions' input and output types.

[1]The term *refinement types* is unfortunately quite overloaded. In this paper, we discuss refinement types in the sense of *subset types* [9].

[2]Informal in the sense that they are lacking machine-checked proofs.

In a blog post [11], Jhala further explains why refinement types are different (and in some aspects, superior to) Hoare logic, with the punchline that "types decompose quantified assertions into quantifier-free refinements", which is a recipe for rendering the logics decidable for SMT-solvers.

The formal connections between refinement types and Hoare logic deserve more systematic studies. In this paper, we present a unifying paradigm—a Hoare logic style formalisation of a refinement type system on a simply-typed λ-calculus with primitive types, products and restricted to first-order functions. Formalising refinement types in the Hoare logic style not only allows us to study the connections between these two systems, it also makes the formalisation easier by avoiding the aforementioned complications in refinement type systems. The formalisation is done in Agda [23, 24], a dependently-typed proof assistant. In our formalisation, shallow Agda terms are used to denote the semantics of the object language and Agda's type system is used as the underlying logic for the refinement predicates.

In a nutshell, we formulate the typing judgement of the refinement type system as $\Gamma\{\phi\} \vdash e : T\{\lambda\nu.\psi\}$. When reading it as a regular typing rule, the typing context is split into two parts: Γ is a list of term variables associated with their base types, and ϕ contains all the refinement predicates about these variables in the context. The base type T and the predicate $\lambda\nu.\psi$ form the refinement type $\{\nu : T \mid \psi\}$ that is assigned to e. On the other hand, if we read the rule as a Hoare triple, e is the program and ϕ and ψ are the pre- and post-conditions of the execution of e under the context Γ.

When we make the analogy between refinement type systems and Hoare logic, another analogy naturally arises. The type analysis with a refinement type system correlates to the weakest precondition calculation in Hoare logic. In fact, the idea of using the weakest precondition for refinement type inference is not new. Knowles and Flanagan [14] have proposed, as future work, calculating the weakest precondition for refinement type reconstruction. In this paper, we explore how to use backward reasoning for typechecking, with our machine-checked formalisation in Agda.

Specifically, this paper presents the following technical contributions:

- We formalise a refinement type system (Section 3 and Section 4) à la Hoare logic and prove, among other meta-theoretical results, that the type system is sound and complete with respect to the denotational semantics (Section 5).
- We define a naïve weakest precondition function wp in lieu of a typechecking algorithm and prove that it is sound and complete with respect to the typing rules (Section 6).
- We revise the formalisation above and present a variant of the refinement type system which preserves the contracts imposed by functions (i.e. λ-abstractions). This requires a more sophisticated weakest precondition function pre and a verification condition generator vc. We again establish the soundness and completeness results of pre and vc (Section 7).

All the formalisation is developed in and checked by Agda (version 2.6.2.1). In fact, the main body of this paper is generated from a literate Agda file that contains all the formal development, including the proofs of all the theorems presented in this paper. The source file of this paper can be obtained online (https://github.com/zilinc/ref-hoare).

2 The Key Idea

Typically, a refinement type can be encoded as a Σ-type in a dependently-typed host language. For example, in Agda's standard library, a refinement type is defined as a record consisting of a value and an irrelevant proof of a property P over the value.[3]

Embedding and working with a dependently-typed language is often quite tedious. Encoding such an object language in another dependently-typed language can typically be done with the help of inductive-recursive techniques [8]. The dependent object language features telescopic contexts in the typing rules. As a result, it poses extra challenges in manipulating typing contexts and in performing type inference, because the dependency induces specific topological orders in solving type constraints.

Realising the connections between refinement types and Hoare logic can be a rescue. When assigning a refinement type to a function (we assume, without loss of generality, that a function only takes one argument), the refinement predicate on the argument asserts the properties that the input possesses, and the predicate on the result type needs to be satisfied by the output. This mimics the structure of a Hoare triple: the predicates on the input and on the output correspond to the pre- and post-conditions respectively.

Another correlation is that, as we have alluded to earlier, in a traditional refinement typing judgement $\overrightarrow{x_i : \tau_i} \vdash e : \tau$ (we use an overhead arrow to denote an ordered vector, and an overhead line for an unordered list; τ_i and τ here are refinement types), the refinement predicates in $\overline{\tau_i}$ correlate to the precondition in a Hoare triple, and the predicate in τ corresponds to the postcondition. Concretely, if each τ_i is in the form $\{\nu : B_i \mid \phi_i\}$, we can take the conjunction of all the ϕ_i to form a proposition about $\overline{x_i}$, which becomes the precondition describing the program state (i.e. the typing context) under which e is executed. Similarly, if τ is $\{\nu : B \mid \psi\}$, then ψ is the postcondition that the evaluation result ν of e must satisfy.

The Hoare triple view of refinement types has many advantages. Firstly, it separates the checking or inference of

[3]https://github.com/agda/agda-stdlib/blob/367e3d6/src/Data/Refinement.agda

the base types and that of the refinement predicates, which is common practice in languages with refinement types (e.g. [14, 31, 33]). The base type information ensures that the refinement predicates are well-typed. Secondly, the separation of types and predicates means that there is no longer any term-dependency in types, and there is no telescopic contexts any more. It makes the formalisation and the reasoning of the system drastically simpler. In particular, the typing contexts no longer need to maintain any particular order.

In this paper, we study a small simply-typed λ-calculus with primitive types, products, and only first-order functions. We assume that all programs are well-typed under the simple type system and only focus on the type refinement. We require all functions (λ-abstractions) to be annotated with refinement types and they are the only places where type annotations are needed. We only typecheck a program against the given annotations, without elaborating the entire syntax tree with refinement types. We reduce the typechecking problem to the computation of the weakest precondition of a program and define a verification condition generator which aggregates all the proof obligations that need to be fulfilled to witness the well-typedness of the program. The proof of the verification conditions is left to the users, who serve as an oracle for solving all logic puzzles. Therefore we do not concern with the decidability of the refinement logic.

3 The Base Language λ^B

Our journey starts with a simply-typed λ-calculus λ^B without any refinement. The syntax of the λ^B is shown in Figure 1. It has ground types of unit ($\mathbb{1}$), Booleans ($\mathbb{2}$) and natural numbers ($\mathbb{N}$), and product types. These types are called base types, meaning that they are the types that can be refined, i.e. they can appear in the base type position B in a typical refinement type $\{\nu : B \mid \phi\}$. The language is restricted to first-order functions by excluding function arrows from the first-class base types. The term language is very standard, consisting of variables (x), constants of the ground types, pairs, projections (π_1 and π_2), function applications (denoted by juxtaposition), if-conditionals, non-recursive local let-bindings, and some arithmetic and logical operators on natural numbers and Booleans.

Although λ-abstractions can only be directly applied, we do not eschew them in the syntax. This allows us to define top-level functions, which can be reused. This design decision is primarily only for practical convenience and is otherwise unimportant. The Agda formalisation follows this design; it handles function and non-function terms separately. Whenever possible, however, we merge the two when we present them in this paper.

Since the typing rule for λ^B is very standard, we directly show its Agda embedding and use it as a tutorial on how we construct the language in Agda. We use an encoding derived from McBride's Kipling language [20], which allows

$$
\begin{array}{llcl}
\text{base types} & B, S, T & ::= & \mathbb{1} \mid \mathbb{2} \mid \mathbb{N} \mid S \times T \\
\text{func. types} & & \ni & S \longrightarrow T \\
\text{expressions} & e & ::= & x \mid () \mid \mathsf{true} \mid \mathsf{false} \\
 & & \mid & \mathsf{ze} \mid \mathsf{su}\ e \\
 & & \mid & (e, e) \mid \pi_1\ e \mid \pi_2\ e \mid f\ e \\
 & & \mid & \textbf{if}\ c\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \\
 & & \mid & \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\
 & & \mid & e_1 \oplus e_2 \\
\text{binary operators} & & \ni & \oplus \\
\text{functions} & f & ::= & \lambda x.e \\
\text{contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : S
\end{array}
$$

Figure 1. Syntax of the language λ^B

```
                         ⟦_⟧τ : U → Set
data U : Set where       ⟦ `1´ ⟧τ = ⊤
  `1´ `2´ `ℕ´ : U        ⟦ `2´ ⟧τ = Bool
  _`×´_ : U → U → U      ⟦ `ℕ´ ⟧τ = ℕ
                         ⟦ S `×´ T ⟧τ = ⟦ S ⟧τ × ⟦ T ⟧τ
```

Figure 2. The Agda definition of the base types and their interpretation

```
data Cx : Set where      ⟦_⟧C : Cx → Set
  `E´ : Cx               ⟦ `E´ ⟧C = ⊤
  _▸_ : Cx → U → Cx      ⟦ Γ ▸ S ⟧C = ⟦ Γ ⟧C × ⟦ S ⟧τ
```

Figure 3. The syntax and the denotation of the typing context

us to index the syntax of the object language with its type in Agda. Therefore, the object term language is type correct by construction, up to simple types.

In Figure 2, we define a universe **U** of codes for base types, and a decoding function $\mathscr{E}[\![\cdot]\!]_{\mathsf{Ty}}$ (⟦_⟧τ in Agda) which maps the syntax to the corresponding Agda types. We do not include a code constructor for function types; a (non-dependent) function type is interpreted according to its input and output types.

McBride [20] uses inductive-recursive definitions [8] for embedding his dependently-typed object language in Agda, which is a pretty standard technique (e.g. [2, 4]). In our base language (and also later with refinement types), since the term-dependency in types has been eliminated by the Hoare logic style formulation, the inductive-recursive definition of the universe à la Tarski and its interpretation is not needed. Nevertheless, we choose to use the vocabulary from that lines of work since the formalisation is heavily inspired by them.

```
data _∋_ : (Γ : Cx)(T : U) → Set where
  top : ∀{Γ}{T} → Γ ▸ T ∋ T
  pop : ∀{Γ}{S T} → Γ ∋ T → Γ ▸ S ∋ T

⟦_⟧∋ : ∀{Γ}{T} → Γ ∋ T → (γ : ⟦ Γ ⟧C) → ⟦ T ⟧τ
⟦ top ⟧∋ (_ , t) = t
⟦ pop i ⟧∋ (γ , _) = ⟦ i ⟧∋ γ
```

Figure 4. Variable indexing in contexts Γ and γ

With the denotation of types, we define what it means for a denotational value to possess a type, following the work on semantic typing [21]. A denotational value v possesses a type T, written $\vDash v : T$, if v is a member of the semantic domain corresponding to the type T. This is to say, v is a shallow Agda value of type $\mathscr{E}[\![T]\!]_{\mathsf{Ty}}$.

Next, we define the typing context for the language λ^B, and the denotation of the context in terms of a nested tuple of shallow Agda values (see Figure 3). The denotation of the typing context gives us a *semantic environment* γ, mapping variables to their denotational values in Agda. A semantic environment γ *respects* the typing context Γ if for all $x \in \mathrm{dom}(\Gamma)$, $\vDash \gamma(x) : \Gamma(x)$.

Variables in λ^B are nameless and are referenced by their de Bruijn indices in the context, with the rightmost (also outermost) element bound most closely. Unlike in Kipling [20], the direction to which the context grows is largely irrelevant, since the context is not telescopic. The variable indexing for the typing context Γ and the semantic context γ are defined in Figure 4 respectively.

Before we continue, we introduce a few combinators that are helpful in simplifying the presentation. ᵏ and ˢ are the **K** and **S** combinators from the SKI calculus. Infix operators ˆ and ˇ are synonyms to the currying and uncurrying functions respectively.

The syntax of the language is defined in Figure 5, together with the interpretation functions ⟦_⟧⊢ and ⟦_⟧⊢⃗. The deep syntax of the object language is indexed by the typing context and the deep type. It therefore guarantees that the deep terms are type correct by construction. There is little surprise in the definition of the typing rules. We only mention that `FUN` has the same type as a normal first-class λ-abstraction does. It can be constructed under any context Γ and does not need to be closed. A function application can be represented equivalently as a let-binding up to simple types, but they have different refinement typing rules, as we will see later in this paper.

The interpretation of the term langauge is entirely standard, mapping object language terms to values of their corresponding Agda types. On paper, we write $\mathscr{E}[\![\cdot]\!]_{\mathsf{Tm}}$ for the denotation function, which takes a deep term and a semantic environment and returns the Agda denotation.

As a simple example, if we want to define a top-level function

$$f_0 : \mathbb{N} \longrightarrow \mathbb{N}$$
$$f_0 = \lambda x. x + 1$$

it can be done in Agda as

```
f₀ : ∀{Γ} → Γ ⊢ `ℕ′ ⟶ `ℕ′
f₀ = FUN (BOP [+] (VAR top) ONE)
```

where `ONE` is defined as `SU ZE`. Note that the function is defined under any context Γ. The denotation of the `f₀` function under any valid semantic environment γ is:

```
⟦f₀⟧⊢ : ∀{Γ}{γ : ⟦ Γ ⟧C} → ℕ → ℕ
⟦f₀⟧⊢ {γ = γ} = ⟦ f₀ ⟧⊢⃗ γ
```

Evaluating this term in Agda results in a λ-term: `λ x → x + 1`, independent of the environment γ.

4 Refinement Typed Language λ^R

We introduce a refinement typed language λ^R that is obtained by equipping λ^B with refinement predicates. We first present the syntax of the language in Figure 6.[4] The upcast operator for non-function expressions is used to make any refinement subtyping explicit in the typing tree.

In contrast to the traditional formulation of refinement type systems, where a typing context is defined as $\overrightarrow{x_i : \tau_i}$, we split it into a base typing context Γ and a refinement predicate ϕ that can be constructed by taking the conjunction of all predicates in $\overline{\tau_i}$. This formulation is arguably more flexible to work with. It does not enforce any ordering in the context entries, and it is easier to add path sensitive constraints that are not bound to a program variable. For example, when typechecking an expression **if** c **then** e_1 **else** e_2, we need to add the constraint c to the context when we zoom in on e_1. This can typically be done by introducing a fresh (ghost) variable x of an arbitrary base type, such as $x : \{\nu : \mathbb{1} \mid c\}$, where $\nu \notin \mathsf{FV}(c)$, and add it to the typing context. In our system, additional conjuncts can be added to the predicate ϕ directly.

Refinement predicates are shallowly embedded as Agda terms of type `Set`. We also define a substitution function in Agda which allows us to substitute the top-most variable in a predicate ϕ with an expression e:

```
_[_]s : ∀{Γ}{T} → (ϕ : ⟦ Γ ▸ T ⟧C → Set) → (e : Γ ⊢ T)
      → (⟦ Γ ⟧C → Set)
ϕ [ e ]s = ^ ϕ ˢ ⟦ e ⟧⊢
```

[4]A remark on the notation: when we talk about the dependent function types in the object languages, we use a slightly longer function arrow ⟶ as a reminder that it is not a first-class type constructor. The typesetting is only subtly different from the normal function arrow → and in fact its semantics overlaps with that of the normal function arrow. In reading and understanding the rules, their difference can usually be dismissed.

```
data _⊢_ (Γ : Cx) : U → Set
data _⊢_⟶_ (Γ : Cx) : U → U → Set

data _⊢_ Γ where
  VAR  : ∀{T} → Γ ∋ T → Γ ⊢ T
  UNIT : Γ ⊢ `1´
  TT   : Γ ⊢ `2´
  FF   : Γ ⊢ `2´
  ZE   : Γ ⊢ `ℕ´
  SU   : Γ ⊢ `ℕ´ → Γ ⊢ `ℕ´
  IF   : ∀{T} → Γ ⊢ `2´ → Γ ⊢ T → Γ ⊢ T → Γ ⊢ T
  LET  : ∀{S T} → Γ ⊢ S → Γ ▸ S ⊢ T → Γ ⊢ T
  PRD  : ∀{S T} → Γ ⊢ S → Γ ⊢ T → Γ ⊢ (S `×´ T)
  FST  : ∀{S T} → Γ ⊢ S `×´ T → Γ ⊢ S
  SND  : ∀{S T} → Γ ⊢ S `×´ T → Γ ⊢ T
  APP  : ∀{S T} → (Γ ⊢ S ⟶ T) → Γ ⊢ S → Γ ⊢ T
  BOP  : (o : ⊕) → Γ ⊢ →⊕ o → Γ ⊢ →⊕ o → Γ ⊢ ⊕→ o

data _⊢_⟶_ Γ where
  FUN : ∀{S T} → Γ ▸ S ⊢ T → Γ ⊢ S ⟶ T
```

```
⟦_⟧⊢ : ∀{Γ}{T} → Γ ⊢ T → ⟦ Γ ⟧C → ⟦ T ⟧τ
⟦_⟧⊢⃗ : ∀{Γ}{S T} → Γ ⊢ S ⟶ T → ⟦ Γ ⟧C → ⟦ S ⟧τ → ⟦ T ⟧τ

--
⟦ VAR x ⟧⊢ = ⟦ x ⟧∋
⟦ UNIT ⟧⊢ = ᵏ tt
⟦ TT ⟧⊢ = ᵏ true
⟦ FF ⟧⊢ = ᵏ false
⟦ ZE ⟧⊢ = ᵏ 0
⟦ SU e ⟧⊢ = ᵏ suc ˢ ⟦ e ⟧⊢
⟦ IF c e₁ e₂ ⟧⊢ = (if_then_else_ ∘ ⟦ c ⟧⊢) ˢ ⟦ e₁ ⟧⊢ ˢ ⟦ e₂ ⟧⊢
⟦ LET e₁ e₂ ⟧⊢ = ^ ⟦ e₂ ⟧⊢ ˢ ⟦ e₁ ⟧⊢
⟦ PRD e₁ e₂ ⟧⊢ = < ⟦ e₁ ⟧⊢ , ⟦ e₂ ⟧⊢ >
⟦ FST e ⟧⊢ = proj₁ ∘ ⟦ e ⟧⊢
⟦ SND e ⟧⊢ = proj₂ ∘ ⟦ e ⟧⊢
⟦ APP f e ⟧⊢ = ⟦ f ⟧⊢⃗ ˢ ⟦ e ⟧⊢
⟦ BOP o e₁ e₂ ⟧⊢ γ = ⟦ e₁ ⟧⊢ γ ⟦ o ⟧⊢⊕ ⟦ e₂ ⟧⊢ γ

--
⟦ FUN e ⟧⊢⃗ = ^ ⟦ e ⟧⊢
```

Figure 5. The Agda embedding of λ^B and the interpretation, side-by-side. ⊕ is the deep syntax for binary operators; →⊕ and ⊕→ return the input and output types of a binary operator respectively. _⟦ o ⟧⊢⊕_ interprets the deep operator o as its Agda counterpart.

ref. types	τ	$::=$	$\{\nu : B \mid \phi\}$	
func. types		$\ni$	$x : \tau \longrightarrow \tau$	(dep. functions)
expressions	$\hat{e}$	$::=$	$\ldots$	(same as λ^B)
		$\mid$	$\hat{e} :: \tau$	(upcast)
functions	$\hat{f}$	$::=$	$\lambda x.\hat{e}$	
ref. contexts	$\hat{\Gamma}$	$::=$	$\Gamma ; \phi$	
predicates	ϕ, ξ, ψ	$::=$	$\ldots$	(a logic of choice)

Figure 6. Syntax for the language λ^R

$$\boxed{\hat{\Gamma}\ \mathsf{wf}}$$

$$\frac{\mathsf{FV}(\phi) \subseteq \mathsf{dom}(\Gamma)}{\Gamma ; \phi\ \mathsf{wf}}\,\textsf{Ctx-Wf}$$

$$\boxed{\Gamma \vdash \{\nu : B \mid \psi\}\ \mathsf{wf}}$$

$$\frac{\mathsf{FV}(\psi) \subseteq \mathsf{dom}(\Gamma) \cup \{\nu\}}{\Gamma \vdash \{\nu : B \mid \psi\}\ \mathsf{wf}}\,\textsf{RefType-Wf}$$

Figure 7. Well-formedness rules for contexts and types

In Figure 7, we show the well-formedness rules for the refinement contexts and for the refinement types. They are checked by Agda's type system and are therefore implicit in the Agda formalisation. The typing rules can be found in Figure 8. Most of the typing rules are straightforward and work in a similar manner to their counterparts in a more traditional formalisation of refinement types. We only elaborate on a few of them.

Variables. The VAR$^{\text{R}}$ rule infers the most precise type—the singleton type—for a variable x. In many other calculi (e.g. [12, 15, 29]), a different variant of the selfification rule is used for variables:[5]

$$\frac{(x : \{\nu : B \mid \phi\}) \in \Gamma_\tau}{\Gamma_\tau \vdash x : \{\nu : B \mid \phi \wedge \nu \equiv x\}}\,\text{Self}$$

We choose not to include the ϕ in the inferred type of x, as such information can be recovered from the typing context via the subtyping rule SUB$^{\text{R}}$.

Constants. For value constants ((), true, false, ze) and function constants ($\oplus$, (,), π_1, π_2, su), the typing rules always infer the exact type for the result. As with the VAR$^{\text{R}}$ rule, we do not carry over the refinement predicates in the premises to the inferred type in the conclusion. Again, no information is lost during this process, as they can be recovered later from the context when needed.

Function applications. The typing rule for function applications is pretty standard. In the work of Knowles and

[5]We use the subscript in Γ_τ to mean the more traditional $\overrightarrow{x_i : \tau_i}$ context, where each τ_i is a refinement type.

$$\boxed{\hat{\Gamma} \vdash_R \hat{e} : \tau}$$

$$\frac{(x : T) \in \Gamma}{\Gamma; \phi \vdash_R x : \{\nu : T \mid \nu = x\}}\,\text{VAR}^{\text{R}}$$

$$\frac{}{\hat{\Gamma} \vdash_R () : \{\nu : \mathbb{1} \mid \nu = ()\}}\,\text{UNIT}^{\text{R}}$$

$$\frac{b \in \{\text{true}, \text{false}\}}{\hat{\Gamma} \vdash_R b : \{\nu : \mathbb{2} \mid \nu = b\}}\,\text{TT}^{\text{R}}/\text{FF}^{\text{R}}$$

$$\frac{}{\hat{\Gamma} \vdash_R \mathsf{ze} : \{\nu : \mathbb{N} \mid \nu = 0\}}\,\text{ZE}^{\text{R}}$$

$$\frac{\hat{\Gamma} \vdash_R \hat{e} : \{\nu : \mathbb{N} \mid \psi\}}{\hat{\Gamma} \vdash_R \mathsf{su}\ \hat{e} : \{\nu : \mathbb{N} \mid \nu = \mathsf{suc}(\hat{e})\}}\,\text{SU}^{\text{R}}$$

$$\frac{\begin{array}{c}\Gamma; \phi \vdash_R \hat{c} : \{\nu : \mathbb{2} \mid \psi'\} \\ \Gamma; \phi \wedge \hat{c} \vdash_R \hat{e}_1 : \{\nu : T \mid \psi\} \qquad \Gamma; \phi \wedge \neg\hat{c} \vdash_R \hat{e}_2 : \{\nu : T \mid \psi\}\end{array}}{\Gamma; \phi \vdash_R \textbf{if}\ \hat{c}\ \textbf{then}\ \hat{e}_1\ \textbf{else}\ \hat{e}_2 : \{\nu : T \mid \psi\}}\,\text{IF}^{\text{R}}$$

$$\frac{\begin{array}{c}\Gamma; \phi \vdash_R \hat{e}_1 : \{x : S \mid \xi\} \qquad \Gamma \vdash \{\nu : T \mid \psi\}\ \text{wf} \\ \Gamma, x : S; \phi \wedge \xi \vdash_R \hat{e}_2 : \{\nu : T \mid \psi\}\end{array}}{\Gamma; \phi \vdash_R \textbf{let}\ x = \hat{e}_1\ \textbf{in}\ \hat{e}_2 : \{\nu : T \mid \psi\}}\,\text{LET}^{\text{R}}$$

$$\frac{\hat{\Gamma} \vdash_R \hat{e}_1 : \{\nu : S \mid \psi_1\} \qquad \hat{\Gamma} \vdash_R \hat{e}_2 : \{\nu : T \mid \psi_2\}}{\hat{\Gamma} \vdash_R (\hat{e}_1, \hat{e}_2) : \{\nu : S \times T \mid \nu = (\hat{e}_1, \hat{e}_2)\}}\,\text{PRD}^{\text{R}}$$

$$\frac{\hat{\Gamma} \vdash_R \hat{e} : \{\nu : T_1 \times T_2 \mid \psi\} \qquad i \in \{1, 2\}}{\hat{\Gamma} \vdash_R \pi_i\ \hat{e} : \{\nu : T_i \mid \nu = \pi_i\ \hat{e}\}}\,\text{FST}^{\text{R}}/\text{SND}^{\text{R}}$$

$$\frac{\begin{array}{c}\Gamma; \phi \vdash_R \hat{f} : x : \{\nu : S \mid \xi\} \longrightarrow \{\nu : T \mid \psi\} \\ x \notin \text{Dom}(\Gamma) \qquad \Gamma; \phi \vdash_R \hat{e} : \{\nu : S \mid \xi\}\end{array}}{\Gamma; \phi \vdash_R \hat{f}\ \hat{e} : \{\nu : T \mid \psi[\hat{e}/x]\}}\,\text{APP}^{\text{R}}$$

$$\frac{\begin{array}{c}\text{ty}(\oplus) = B_1 \to B_1 \to B_2 \\ \hat{\Gamma} \vdash_R \hat{e}_1 : \{\nu : B_1 \mid \psi_1\} \qquad \hat{\Gamma} \vdash_R \hat{e}_2 : \{\nu : B_1 \mid \psi_2\}\end{array}}{\hat{\Gamma} \vdash_R \hat{e}_1 \oplus \hat{e}_2 : \{\nu : B_2 \mid \nu = \hat{e}_1 \oplus \hat{e}_2\}}\,\text{BOP}^{\text{R}}$$

$$\frac{\Gamma; \phi \vdash_R \hat{e} : \{\nu : S \mid \psi\} \qquad \Gamma, x : S; \phi \vDash \psi \Rightarrow \psi'}{\Gamma; \phi \vdash_R \hat{e} :: \{\nu : S \mid \psi'\}}\,\text{SUB}^{\text{R}}$$

$$\frac{\Gamma; \phi \vdash_R \hat{e} : \{\nu : S \mid \psi\} \qquad \Gamma \vDash \phi' \Rightarrow \phi}{\Gamma; \phi' \vdash_R \hat{e} : \{\nu : S \mid \psi\}}\,\text{WEAK}^{\text{R}}$$

$$\boxed{\hat{\Gamma} \vdash_R \hat{f} : x : \tau_1 \longrightarrow \tau_2}$$

$$\frac{\Gamma, x : S; \xi \vdash_R \hat{e} : \{\nu : T \mid \psi\}}{\Gamma; \phi \vdash_R \lambda x.e : x : \{\nu : S \mid \xi\} \longrightarrow \{\nu : T \mid \psi\}}\,\text{FUN}^{\text{R}}$$

Figure 8. Static semantics of λ^R

Flanagan [15], a compositional version of this rule is used instead. To summarise the idea, consider the typical function application rule, which has the following form:

$$\frac{\Gamma_\tau \vdash f : (x : \tau_1) \to \tau_2 \qquad \Gamma_\tau \vdash e : \tau_1' \qquad \Gamma_\tau \vdash \tau_1' \sqsubseteq \tau_1}{\Gamma_\tau \vdash f\ e : \tau_2[e/x]}\,\text{App}$$

In the refinement in the conclusion, the term e is substituted for x. This would circumvent the type abstraction τ_1' of e, exposing the implementation details of the argument to the resulting refinement type $\tau_2[e/x]$. It also renders the type $\tau_2[e/x]$ arbitrarily large due to the presence of e. To rectify this problem, Knowles and Flanagan [15] propose the result type to be existential: $\exists x : \tau_1'. \tau_2$. Which application rule to include largely depends on the language design. We use the traditional one here and the compositional one later in this paper to contrast the two.

Jhala and Vazou [12] stick to the regular function application rule, but with some extra restrictions. They require the function argument to be in A-normal form (ANF) [34], i.e. the argument being a variable instead of an arbitrary expression. This reduces the load on the SMT-solver and helps them remain decidable in the refinement logic. We do not need the ANF restriction in our system for decidability, and the argument term will always be fully reduced in Agda while conducting the meta-theoretical proofs.

Let-bindings. In the LET^{R} rule, the well-formedness condition $\Gamma \vdash \{\nu : T \mid \psi\}$ wf implies that ψ does not mention the locally-bound variable x, preventing the local binder from creeping into the resulting type of the let-expression. The let-binding and the function application rule give similar power in reasoning, thanks to the SUB^{R} rule. The structure of the proofs are slightly different though because the SUB^{R} nodes need to be placed at different positions.

Subtyping and weakening. Key to a refinement type system is the subtyping relation between types. Typically, the (partly syntactic) subtyping judgement looks like:

$$\frac{\Gamma_\tau, x : B \vDash \phi \Rightarrow \phi'}{\Gamma_\tau \vdash \{\nu : B \mid \phi\} \sqsubseteq \{\nu : B \mid \phi'\}}\,\text{Sub}$$

$$\frac{\Gamma_\tau \vdash \sigma_2 \sqsubseteq \sigma_1 \qquad \Gamma_\tau, x : \sigma_2 \vdash \tau_1 \sqsubseteq \tau_2}{\Gamma_\tau \vdash x : \sigma_1 \to \tau_1 \sqsubseteq x : \sigma_2 \to \tau_2}\,\text{Sub-Fun}$$

In our language, since we only support first-order functions, the subtyping rule for functions is not needed. It can be achieved by promoting the argument and the result of a function application separately. Since function types are excluded from the universe **U**, subtyping can be defined on the entire domain of **U**, and in a fully semantic manner. We directly define the subtyping-style rules (SUB^{R}, WEAK^{R}) in terms of a logical entailment: $\phi \vDash \psi \Rightarrow \psi' \overset{def}{=} \forall \gamma\ x. \phi\ \gamma \wedge \psi\ (\gamma, x) \to \psi'\ (\gamma, x)$.

If we allowed refinement predicates over function spaces, it would require a full semantic subtyping relation that also works on the function space. This has been shown to be possible, e.g. interpreting the types in a set-theoretic fashion as in Castagna and Frisch [1]'s work. It is however far from trivial to encode a set theory that can be used for the interpretation of functions in Agda's type system (e.g. [16] is an attempt to define Zermelo–Fraenkel set theory **ZF** in Agda).

The subtyping rule (SUB^{R}) and the weakening rule (WEAK^{R}) roughly correspond to the left- and right- consequence rules of Hoare logic respectively. All the subtyping in our system is explicit. For instance, unlike rule App above, in order to apply a function, we have to explicitly promote the argument with a SUB^{R} node, if its type is not already matching the argument type expected by the function. As a notational convenience, in the typing rules we write $\hat{\Gamma} \vdash_R \hat{e} :: \tau$ to mean $\hat{\Gamma} \vdash_R \hat{e} :: \tau : \tau$, as the inferred type is always identical to the one that is promoted to.

Comparing the SUB^{R} rule with the right-consequence rule in Hoare logic, which reads

$$\frac{\{P\}\, s\, \{Q\} \qquad Q \rightarrow Q'}{\{P\}\, s\, \{Q'\}}\text{Cons-R}$$

we can notice that in the SUB^{R} rule, the implication says $\phi \vDash \psi \Rightarrow \psi'$. In Cons-R, in contrast, the precondition P is not involved in the implication. This is because of the nature of the underlying language. In an imperative language to which the Hoare logic is applied, P and Q are predicates over the program states. A variable assignment statement or reference update operation will change the state. Therefore after the execution of the statement s, the predicate P is no longer true and all the relevant information are stored in Q. In our purely functional language λ^R, ϕ is a predicate over the typing context Γ, and a typing judgement does not invalidate ϕ. Moreover, in practice, when assigning a refinement type to an expression, the refinement predicate typically only concerns the term being typed, and does not talk about variables in Γ that are not directly relevant. Therefore it is technically possible not to require ϕ in the implication, but it renders the system more cumbersome to use.

As for weakening, in contrast to the more canonical structural rule [17]:

$$\frac{\vdash \Gamma_{\tau_1}, \Gamma_{\tau_2}, \Gamma_{\tau_3} \qquad \Gamma_{\tau_1}, \Gamma_{\tau_3} \vdash e : \tau}{\Gamma_{\tau_1}, \Gamma_{\tau_2}, \Gamma_{\tau_3} \vdash e : \tau}$$

the WEAK^{R} rule only changes the predicates in the context and does not allow for adding new binders to the simply-typed context Γ. It compares favourably to those with a more syntactic refinement-typing context. For a traditional refinement-typing context $\overrightarrow{x_i : \tau_i}$, if the weakening lemma is to be defined in a general enough manner to allow weakening to happen arbitrarily in the middle of the context, some tactics will be required to syntactically rearrange the context to make the weakening rule applicable. Our weakening rule is purely semantic and therefore does not require syntactic rewriting before it can be applied.

Functions. The FUN^{R} rule can be used to construct a λ-abstraction under any context Γ and the λ-term does not need to be closed. The function body $\hat{e}$ is typed under the extended context $\Gamma, x : S$, but the predicate part does not include ϕ. This does not cause any problems because ξ is itself a predicate over the context and the function argument, and also if more information about the context needs to be drawn, it can be done via the SUB^{R} rule at a later stage.

The pen-and-paper formalisation above leaves some aspects informal for presentation purposes. One instance of the discrepancy is that, when we type the term $\hat{e}_1 + \hat{e}_2$, the resulting predicate is $\nu = \hat{e}_1 + \hat{e}_2$. What has been implicit in the rules is the reflection of object terms into the logical system.

In our formal development, the underlying logical system is Agda's type system, therefore we want to reflect the refinement-typed term language into the Agda land. We do it as a two-step process: we first map the refinement-typed language to the simply-typed language by erasure, and then reflect the simply-typed terms into logic using the already-defined interpretation function $\mathscr{E}[\![\cdot]\!]_{\text{Tm}}$, with which we interpret the object language as Agda terms.

The erasure function $\ulcorner\cdot\urcorner^{\text{R}}$ removes all refinement type information from a refinement-typed term (also, typing tree) and returns a corresponding simply-typed term (also, typing tree). Essentially, the erasure function removes the refinement predicates, and any explicit upcast (SUB^{R}) nodes from the typing tree.

Now we can define the deep syntax of the λ^R language along with its typing rules in Agda. When an expression $\hat{e}$ in the object language has an Agda type `Γ { ϕ }⊢ T { ψ }`, it means that under context Γ which satisfies the precondition ϕ, the expression $\hat{e}$ can be assigned a refinement type $\{\nu : T \mid \psi\}$. For functions, we have `Γ { ϕ }⊢ S { ξ }⟶ T { ψ }` which keeps track of the predicates on the context Γ, on the argument and on the result respectively. The datatypes and the erasure function $\ulcorner\cdot\urcorner^{\text{R}}$ are inductive-recursively defined. For space reasons, we omit the Agda definitions from the paper. They can be found in the Agda source file of the paper.

The context weakening rule WEAK^{R} in Figure 8 is in fact admissible in our system, therefore it is not included as a primitive construct in the formal definition of the language.

Lemma 4.1 (Weakening is admissible)**.** *For any typing tree $\Gamma; \phi \vdash_R \hat{e} : \tau$, if $\phi' \Rightarrow \phi$ under any semantic environment γ that respects Γ, then there exists a typing tree with the stronger context $\Gamma; \phi'$, such that $\Gamma; \phi' \vdash_R \hat{e}' : \tau$ and $\ulcorner\hat{e}\urcorner^R = \ulcorner\hat{e}'\urcorner^R$.*

Continuing on the `f₀` function defined in Section 3, if we want to lift it to a function definition in λ^R, we will need to add refinements and insert explicit upcast nodes:

$$\begin{aligned} f_0^{\text{R}} &: (x : \{\nu : \mathbb{N} \mid \nu < 2\}) \longrightarrow \{\nu : \mathbb{N} \mid \nu < 4\} \\ f_0^{\text{R}} &= \lambda x. (x + 1 :: \{\nu : \mathbb{N} \mid \nu < 4\}) \end{aligned}$$

In Agda, it is defined as follows:

```
f₀ᴿ : `E´ { ᵏ ⊤ }⊢ `ℕ´ { (_< 2) ∘ proj₂ }⟶
                   `ℕ´ { (_< 4) ∘ proj₂ }
f₀ᴿ = FUNᴿ (SUBᴿ (BOPᴿ [+] (VARᴿ top) ONEᴿ) _ prf)
```

The upcast node `SUBᴿ` needs to be accompanied by an evidence (i.e. an Agda proof term `prf` whose definition is omitted) to witness the semantic entailment $x < 2 \vDash \nu = x + 1 \Rightarrow \nu < 4$.

To demonstrate the function application of `f₀ᴿ`, we define the following expression:

$$ex^{\mathrm{R}}_0 : \{\nu : \mathbb{N} \mid \nu < 5\}$$
$$ex^{\mathrm{R}}_0 = f^{\mathrm{R}}_0 \; (1 :: \{\nu : \mathbb{N} \mid \nu < 2\}) :: \{\nu : \mathbb{N} \mid \nu < 5\}$$

The inner upcast is for promoting the argument 1, which is inferred the exact type $\{\nu : \mathbb{N} \mid \nu = 1\}$, to match f^{R}_0's argument type. The outer upcast is for promoting the result of the application from $\{\nu : \mathbb{N} \mid \nu < 4\}$ to $\{\nu : \mathbb{N} \mid \nu < 5\}$. In Agda, two proof terms need to be constructed for the upcast nodes in order to show that the argument and the result of the application are both type correct:

```
ex₀ᴿ : `E′ { ᵏ ⊤ }⊢ `ℕ′ { (_< 5) ∘ proj₂ }
ex₀ᴿ = SUBᴿ (APPᴿ {ψ = (_< 4) ∘ proj₂} f₀ᴿ
                (SUBᴿ ONEᴿ _ λ _ _ _ → s≡1⇒s<2))
            _ λ _ _ _ → t<4⇒t<5
```

5 Meta-Properties of λ^R

Instead of proving the textbook type soundness theorems (progress and preservation) [10, 38] that rest upon subject reduction, we instead get for free the semantic type soundness theorem à la Milner [21] for the base language λ^B because of the way the term language is embedded and interpreted in Agda.

Theorem 5.1 (Semantic soundness). *If $\Gamma \vdash e : T$ and the semantic environment γ respects the typing environment Γ, then $\vDash \mathscr{E}[\![e]\!]_{Tm}\gamma : T$.*

We take the same semantic approach to type soundness and establish the the refinement soundness theorem for λ^R. We use the notation $\phi \vDash \psi$ for the semantic entailment relation in the underlying logic, which, in our case, is Agda's type system. To relate a semantic environment γ to a refinement typing context $\hat{\Gamma}$, we proceed with the following definitions.

Definition 5.2. A semantic environment γ satisfies a predicate ϕ, if $\mathsf{FV}(\phi) \subseteq \mathrm{dom}(\gamma)$ and $\varnothing \vDash \phi\ \gamma$. We write $\phi\ \gamma$ to mean $\phi[\gamma(x_i)/x_i]$ for all free variables x_i in ϕ.

Definition 5.3. A semantic environment γ respects a refinement typing context $\Gamma; \phi$ if γ respects Γ and satisfies ϕ.

We define what it means for a denotational value to possess a refinement type, and extend the notion of semantic typing to refinement types.

Definition 5.4. A value v posesses a refinement type $\{\nu : T \mid \psi\}$, written $\vDash v : \{\nu : T \mid \psi\}$, if $\vDash v : T$ and $\varnothing \vDash \psi[v/\nu]$.

Definition 5.5 (Refinement semantic typing). $\hat{\Gamma} \vDash \hat{e} : \tau$ if $\vDash \mathscr{E}[\![\ulcorner \hat{e} \urcorner^{\mathrm{R}}]\!]_{\mathrm{Tm}}\gamma : \tau$ for all γ that respects $\hat{\Gamma}$.

With the notion of refinement semantic typing, we can state the refinement (semantic) type soundness theorem as follows.

Theorem 5.6 (Refinement soundness). *If $\hat{\Gamma} \vdash_R \hat{e} : \tau$ then $\hat{\Gamma} \vDash e : \tau$.*

The converse of this theorem is also true. It states the completeness of our refinement type system with respect to semantic typing.

Theorem 5.7 (Refinement completeness). *If $\Gamma \vdash e : T$ and for all semantic context γ that respects $\hat{\Gamma}$, $\vDash \mathscr{E}[\![e]\!]_{Tm}\gamma : \{\nu : T \mid \psi\}$, then there exists a refinement typing $\hat{\Gamma} \vdash_R \hat{e} : \{\nu : T \mid \psi\}$ such that $\ulcorner \hat{e} \urcorner^R = e$.*

Note that for the completeness theorem, since we only need to construct one such refinement typed expression (or equivalently, typing tree), the proof is not unique, in light of the SUBᴿ and WEAKᴿ rules.

With the refinement soundness and completeness theorems, we can deduce a few direct but useful corollaries.

Corollary 5.8. *Refinement soundness holds for closed terms.*

Corollary 5.9. *For refinement typing judgements, the predicate ϕ over the context is an invariant, namely, $\Gamma; \phi \vdash_R \hat{e} : \{\nu : T \mid \lambda\nu.\, \phi\}$.*

Corollary 5.10 (Consistency). *It is impossible to assign a void refinement type to an expression $\hat{\Gamma} \vdash_R \hat{e} : \{\nu : T \mid \mathsf{false}\}$, if there exists a semantic environment γ that respects $\hat{\Gamma}$.*

6 Typechecking λ^R by Weakest Precondition

A naïve typechecking algorithm can be given to the λ^R language, in terms of the weakest precondition predicate transformer [6]. Since all functions in λ^R are required to be annotated with types, it is possible to assume the postcondition of a function and compute the weakest precondition. If the given precondition in the type signature entails the weakest precondition, we know that the program is well-typed according to the specification (i.e. the type signature).

In an imperative language, when a variable x gets assigned, the Hoare triple is $\{Q[e/x]\}\ x := e\ \{Q\}$, which means that the weakest precondition can be obtained by simply substituting the variable x in Q by the expression e. The Hoare logic style typing judgement $\Gamma\{\phi\} \vdash e : T\{\psi\}$ in our purely functional language can be considered as assigning the value of e to a fresh variable ν. Therefore the weakest precondition function wp ψ e can be defined analogously as a substitution of the top binder ν in ψ with the value of e, resulting in a predicate over a semantic environment γ that respects Γ.

```
wp : ∀{Γ}{T} → (⟦ Γ ▸ T ⟧C → Set) → Γ ⊢ T → (⟦ Γ ⟧C → Set)
wp ψ e = ψ [ e ]s
```

The completeness and soundness of the wp function with respect to the typing rules of λ^R are direct corollaries of the refinement soundness and completeness theorems (Theorem 5.6 and Theorem 5.7) respectively.

Theorem 6.1 (Completeness of wp w.r.t. λ^R typing). *If* $\Gamma; \phi \vdash_R \hat{e} : \{\nu : T \mid \psi\}$*, then* $\phi\ \gamma \Rightarrow wp\ \psi\ \ulcorner \hat{e} \urcorner^R\ \gamma$ *for any semantic environment* γ *that respects* Γ*.*

Theorem 6.2 (Soundness of wp w.r.t. λ^R typing). *For an expression* $\Gamma \vdash e : T$ *in* λ^B *and a predicate* ψ*, there must exist a type derivation* $\Gamma; wp\ \psi\ e \vdash_R \hat{e} : \{\nu : T \mid \psi\}$ *such that* $\ulcorner \hat{e} \urcorner^R = e$*.*

The wp function checks that, when a type signature is given to an expression *e*, it can infer the weakest precondition under which *e* is typeable. Writing in natural deduction style, the algorithmic typing rule looks like:

$$\frac{\cdots}{\Gamma; \phi \mapsto e : \{\nu : T \mid \psi\}}$$

Contrary to regular algorithmic typing rules (e.g. in bidirectional typing [7]), where the context and the expression are typically inputs, and the type is either input or output depending on whether it performs type checking or synthesise, in our formulation, the expression and the type are the inputs and (the predicate part of) the context is the output.

The wp function only checks whether an expression is well-typed by inferring the weakest context, but it does not elaborate the typing tree by annotating each sub-expression with a type, nor does it automatically construct proof terms. Despite the limitation, this method can still be applied to program verification tasks in which the exact refinement typing tree need not be constructed, or when the automatic construction of proof terms is not required. For instance, we intend to augment the Cogent language [26, 27, 30], a purely functional language for the ease of formal verification of systems code, with refinement types. In Cogent's verification framework, a fully elaborated refinement typing tree will not be necessary, and the functional correctness of a system is manually proved in Isabelle/HOL. Since proof engineers are already engaged, we do not have to rely on an SMT-solver to fully construct the proof objects. It is a bonus, but not a prerequisite.

7 Function Contracts with λ^C

As we have seen, wp is easy to define and works uniformly across all terms. Yet it has a very unfortunate drawback—it is defined on the simply-typed language and is oblivious to the function signatures, and consequently does not preserve function contracts. This is however not the only problem. In the definition of wp, when it is applied to an expression containing function applications, instead of β-reducing the applications, it should inspect the function's type signature and produce verification conditions for the function contracts. To this end, the denotation function $\mathscr{E}[\![\cdot]\!]_{\mathsf{Tm}}$ and subsequently the refinement typing rules also need to be revised.

We define a variant of the language λ^C, in which the function contracts are respected.[6] It is worth mentioning that the language is not yet compositional in the sense of [15], as the weakest precondition computation still draws information from the implementation of expressions, which we will see later in this section.

7.1 The λ^C Language

The syntax of λ^C is the same as λ^R, and its typing rules are very similar to those of λ^R as well. Despite the fact that the λ^C language will reflect its term language to Agda in a slightly different way (more details in Section 7.2), we only make two changes in the typing rules for λ^C:

$$\boxed{\hat{\Gamma} \vdash_C \hat{e} : \tau}$$

$$\frac{\Gamma; \phi \vdash_C \hat{e}_1 : \{x : S \mid \xi\} \qquad \Gamma \vdash \{\nu : T \mid \psi\}\ \mathsf{wf} \qquad \Gamma, x : S; \phi \wedge x = \hat{e}_1 \vdash_C \hat{e}_2 : \{\nu : T \mid \psi\}}{\Gamma; \phi \vdash_C \textbf{let}\ x = \hat{e}_1\ \textbf{in}\ \hat{e}_2 : \{\nu : T \mid \psi\}}\ \mathsf{LET}^{\mathsf{C}}$$

$$\frac{\Gamma; \phi \vdash_C \hat{f} : x : \{\nu : S \mid \xi\} \longrightarrow \{\nu : T \mid \psi\} \qquad x \notin \mathsf{Dom}(\Gamma) \qquad \Gamma; \phi \vdash_C \hat{e} : \{\nu : S \mid \xi\}}{\Gamma; \phi \vdash_C \hat{f}\ \hat{e} : \{\nu : T \mid \exists x : \xi[x/\nu].\psi\}}\ \mathsf{APP}^{\mathsf{C}}$$

As suggested by Knowles and Flanagan [15], the result of a function application can be made existential for retaining the abstraction over the function's argument. This idea is implemented as the rule APP$^{\mathsf{C}}$. The choice of using this favour of function application is purely incidental—offering a contrast to the other variant used in λ^R. In practice, we believe both rules have their place in a system. The existential version is significantly limited in the conclusions that it can lead to, and renders some basic functions useless. For instance, we define an inc function as follows:

$$\mathsf{inc} : (x : \mathbb{N}) \longrightarrow \{\nu : \mathbb{N} \mid \nu = x + 1\}$$
$$\mathsf{inc} = \lambda x. x + 1$$

The function's output is already giving the exact type of the result. With the APP$^{\mathsf{C}}$ rule, we cannot deduce that inc 0 is 1, which is intuitively very obvious. In fact, if the input type of inc is kept unrefined, then we can hardly draw any conclusion about the result of this function. This behaviour can be problematic when users define, say, arithmetic operations as functions.

The LET$^{\mathsf{C}}$ rule differs from LET$^{\mathsf{R}}$ in a way that the precondition of the expression e_2 is ϕ in conjunction with the exact refinement $x = \hat{e}_1$ for the new binder x instead of the arbitrary postcondition ξ of e_1. Intuitively, because the exact type of the local binder is added to the context when type-checking e_2, when we compute the weakest precondition of the let-expression (later in Figure 9), we can assume the trivial postcondition $\lambda\_.\,\mathsf{true}$ of e_1. This makes the LET$^{\mathsf{C}}$ rule significantly easier to work with. Unfortunately, we do not yet have formal evidence to conclude with full confidence

[6]The superscript C in λ^C means "contract".

whether the LET$^{\text{R}}$ rule can be used in this system instead of LET$^{\text{C}}$ or not. Also note that, LET$^{\text{C}}$ gives different reasoning power than APP$^{\text{C}}$ does, and they nicely complement each other in the system.

7.2 Annotated Base Language λ^A

To typecheck of λ^C, we define λ^A, a variant of the base language λ^B. It differs in that the functions are accompanied by type signatures. We denote function expressions in λ^A as $f :: (x : \xi) \longrightarrow \psi$, instead of a bare unrefined f.

To establish the connection between λ^C and λ^A, an erasure function $\ulcorner\cdot\urcorner^{\text{C}}$ is defined, taking a λ^C term to the corresponding λ^A term. It preserves the function's type annotations in λ^C, so that we know that when a λ^A term is typed, the functions are typed in accordance with their type signatures.

One reason why λ^R does not preserve the function contracts is because the way we interpret function calls. Imagine an expression $((f :: (x : \xi) \longrightarrow \psi)\ 1) + 2$ in the language λ^C where $f \equiv \lambda x.\, x + 1$, $\xi \equiv \lambda x.\, x < 2$ and $\psi \equiv \lambda \nu.\, \nu < 4$, which is well-typed. Ideally, the only knowledge that we can learn about the function application should be drawn from its type signature, namely $\lambda \nu.\, \nu < 4$ here. Therefore, the most precise type we can assign to the whole expression is $\{\nu : \mathbb{N} \mid \nu < 6\}$. However, according to the typing rule ADD$^{\text{C}}$, the inferred refinement predicate of the result of the addition will be an Agda term $\mathscr{E}[\![f\ 1]\!]_{\mathsf{Tm}}\gamma + \mathscr{E}[\![2]\!]_{\mathsf{Tm}}\gamma$ for any γ. As the predicate reduces in Agda, it means that we can in fact conclude that the result is equal to 4, which is more precise than what the function contract tells us—we again lost the abstraction over f.

To fix the problem, we revise the definition of $\mathscr{E}[\![\cdot]\!]_{\mathsf{Tm}}$. Instead of interpreting functions as their Agda shallow embedding, we `postulate` the interpretation of functions as δ:

```
    postulate
      δ : ∀{Γ}{S T}{ξ}{ψ} → Γ ⊢ᴬ S { ξ }⟶ T { ψ }
                          → ⟦ Γ ⟧C → ⟦ S ⟧τ → ⟦ T ⟧τ
```

It allows us to reflect functions in the object language into the logic as uninterpreted functions. In the example above, it will stop the shallow postcondition from reducing to 4, retaining a symbolic representation of the function f. We define a new interpretation function $\mathscr{E}[\![\cdot]\!]^{\mathsf{A}}_{\mathsf{Tm}}$ for λ^A terms (`⟦_⟧⊢ᴬ` and `⟦_⟧⊢ᴬ⃗` in the Agda formalisation). It is defined in the same way as $\mathscr{E}[\![\cdot]\!]_{\mathsf{Tm}}$, with the exception of functions:

```
                    ⟦ f ⟧⊢ᴬ⃗ = δ f
```

On the other hand, when we type any expressions in the language λ^C, we need to add the known function contracts to the precondition ϕ. The function contract can be extracted automatically by a `mkC` function defined as follows:

```
mkC : ∀{Γ}{S T}{ξ}{ψ} → Γ ⊢ᴬ S { ξ }⟶ T { ψ } → Set
mkC {Γ = Γ}{S = S}{ξ = ξ}{ψ = ψ} f =
  {γ : ⟦ Γ ⟧C} → (x : ⟦ S ⟧τ) → ξ (γ , x) → ψ ((γ , x) , δ f γ x)
```

7.3 Typechecking λ^A

In order to typecheck λ^A, which is a language that is already well-typed with respect to simple types, and all functions are annotated with refinement types, we want to have a similar deterministic procedure as we had in Section 6. Unfortunately, in the presence of the function boundaries, the weakest precondition computation cannot be done simply by substituting in the expressions.

We borrow the ideas from computing weakest preconditions for imperative languages with loops. Specifically, we follow the development found in Nipkow and Klein [22, §12.4]'s book. In standard Hoare logic, it is widely known that the loop-invariant for a WHILE-loop cannot be computed using the weakest precondition function wp [6], as the function is recursive and may not terminate. In Nipkow and Klein [22]'s work, for Isabelle/HOL to deterministically generate the verification condition for a Hoare triple, it requires the users to provide annotations for loop-invariants. It then divides the standard wp function into two functions: pre and vc. The former computes the weakest precondition nearly as wp, except that in the case of a WHILE-loop, it returns the annotated invariant immediately. The latter then checks that the provided invariants indeed make sense. Intuitively, for a WHILE-loop, it checks that the invariant I together with the loop condition implies the precondition of the loop body, which needs to preserve I afterwards, and that I together with the negation of the loop-condition implies the postcondition. In all other cases, the vc function simply recurses down the sub-statements and aggregates verification conditions.

Although there is no recursion—the functional counterparts to loops of an imperative language—in our language, the situation with functions is somewhat similar to WHILE-loops. We also cannot compute the weakest precondition according to the expressions, but have to rely on user annotations, for a different reason. We can also divide the wp computation into pre and vc. The function pre immediately returns the precondition of a function, which is the refinement predicate of the argument type. Then vc additionally validates the provided function signatures. In particular, we need to check that in a function application: (1) the function's actual argument is of a supertype to the prescribed input type; (2) the function's prescribed output type implies the postcondition of the function application inferred from the program context. Additionally, vc needs to recurse down the syntax tree and gather verification conditions from subexpressions, and, in particular, descend into the function definition to check that it meets the given type signature. The definitions of the pre and the vc functions are shown in Figure 9 and Figure 10 respectively.[7]

Unlike the development in the book of Nipkow and Klein [22], in our language λ^A, the definition of pre deviates from

[7] $\cap$ is the intersection of predicates defined in Agda's standard library as: $P \cap Q = \lambda \gamma \rightarrow P\,\gamma \times Q\,\gamma$.

```
pre  : ∀{Γ}{T}(ψ : ⟦ Γ ▸ T ⟧C → Set) → (e : Γ ⊢ᴬ T)
      → (⟦ Γ ⟧C → Set)
pre⃗ : ∀{Γ}{S T}{ξ}{ψ} → Γ ⊢ᴬ S { ξ }⟶ T { ψ }
      → (⟦ Γ ▸ S ⟧C → Set)

pre ψ (SUᴬ e) = pre (ᵏ ⊤) e ∩ ψ [ SUᴬ e ]sᶜ
pre ψ (IFᴬ c e₁ e₂)
  = pre (ᵏ ⊤) c
  ∩ (if_then_else_ ∘ ⟦ c ⟧⊢ᴬ) ˢ pre ψ e₁ ˢ pre ψ e₂
pre ψ (LETᴬ e₁ e₂)
  = pre (ᵏ ⊤) e₁
  ∩ ^ (pre (λ ((γ , _) , t) → ψ (γ , t)) e₂) ˢ ⟦ e₁ ⟧⊢ᴬ
pre ψ (PRDᴬ e₁ e₂)
  = pre (ᵏ ⊤) e₁ ∩ pre (ᵏ ⊤) e₂ ∩ ψ [ PRDᴬ e₁ e₂ ]sᶜ
pre ψ (FSTᴬ e) = pre (ᵏ ⊤) e ∩ ψ [ FSTᴬ e ]sᶜ
pre ψ (SNDᴬ e) = pre (ᵏ ⊤) e ∩ ψ [ SNDᴬ e ]sᶜ
pre _ (APPᴬ {ξ = ξ}{ψ = ψ} f e) = pre ξ e
pre ψ (BOPᴬ o e₁ e₂)
  = pre (ᵏ ⊤) e₁ ∩ pre (ᵏ ⊤) e₂ ∩ ψ [ BOPᴬ o e₁ e₂ ]sᶜ
pre ψ e = ψ [ e ]sᶜ -- It's just subst for the rest

pre⃗ {ξ = ξ}{ψ = ψ} (FUNᴬ e) = ξ ∩ pre ψ e
```

Figure 9. The Agda definition of pre.

wp by quite a long way. For example, the typing rule for su looks like:

$$\frac{\Gamma; \phi \vdash_C : \hat{e} : \{\nu : \mathbb{N} \mid \xi\}}{\Gamma; \phi \vdash_C : \text{su } \hat{e} : \{\nu : \mathbb{N} \mid \nu = \text{suc } \hat{e}\}} \text{SU}^{\text{C}}$$

Intuitively, when we run the wp backwards on su $\hat{e}$ with a postcondition ψ, it results in $\psi[\mathscr{E}[\![\text{suc } \hat{e}]\!]^{\mathsf{A}}_{\mathsf{Tm}}\gamma/\nu]$ for a semantic environement γ. The inferred refinement ξ of $\hat{e}$ in the premise is arbitrary and appears to be irrelevant to the computation of the weakest precondition of the whole term. Therefore we can set ξ to be the trivial refinement (true) and there is nothing to be assumed about the context to refine $\hat{e}$. This is however not the case in the presence of function contracts. In general, a trivial postcondition does not entail a trivial precondition: pre ϕ ($\lambda\_$. true) $\hat{e} \neq$ ($\lambda\_$. true). For instance, if $\hat{e}$ is a function application, then we also need to compute the weakest precondition for the argument to satisfy the contract.

Our vc function also differs slightly from its counterpart in the imperative setting: it additionally takes the precondition as an argument. This is because in a purely functional language, we do not carry over all the information in the precondition to the postcondition, as the precondition is an invariant (recall that in the subtyping rule SUB$^{\text{R}}$, the entailment is $\phi \vDash \psi \Rightarrow \psi'$).

```
vc  : ∀{Γ}{T} → (⟦ Γ ⟧C → Set) → (⟦ Γ ▸ T ⟧C → Set)
      → Γ ⊢ᴬ T → Set
vc⃗ : ∀{Γ}{S T}{ξ}{ψ} → (⟦ Γ ⟧C → Set)
      → Γ ⊢ᴬ S { ξ }⟶ T { ψ } → Set

vc ϕ ψ (SUᴬ e) = vc ϕ (ᵏ ⊤) e
vc ϕ ψ (IFᴬ c e₁ e₂)
  = vc ϕ (ᵏ ⊤) c
  × vc (λ γ → ϕ γ × ⟦ c ⟧⊢ᴬ γ ≡ true) ψ e₁
  × vc (λ γ → ϕ γ × ⟦ c ⟧⊢ᴬ γ ≡ false) ψ e₂
vc ϕ ψ (LETᴬ e₁ e₂)
  = vc ϕ (ᵏ ⊤) e₁
  × vc (λ (γ , s) → ϕ γ × s ≡ ⟦ e₁ ⟧⊢ᴬ γ)
       (λ ((γ , _) , t) → ψ (γ , t)) e₂
vc ϕ ψ (PRDᴬ e₁ e₂) = vc ϕ (ᵏ ⊤) e₁ × vc ϕ (ᵏ ⊤) e₂
vc ϕ ψ (FSTᴬ e) = vc ϕ (ᵏ ⊤) e
vc ϕ ψ (SNDᴬ e) = vc ϕ (ᵏ ⊤) e
vc {Γ} ϕ ψ´ (APPᴬ {S = S}{T = T}{ξ = ξ}{ψ = ψ} f e)
  = vc⃗ ϕ f
  × vc ϕ ξ e
  × (∀(γ : ⟦ Γ ⟧C)(s : ⟦ S ⟧τ)(t : ⟦ T ⟧τ)
    → ϕ γ → ξ (γ , s) → ψ ((γ , s) , t) → ψ´ (γ , t))
vc ϕ ψ (BOPᴬ o e₁ e₂) = vc ϕ (ᵏ ⊤) e₁ × vc ϕ (ᵏ ⊤) e₂
vc _ _ _ = ⊤

vc⃗ {Γ = Γ}{S = S}{T = T} ϕ (FUNᴬ {ξ = ξ}{ψ = ψ} e)
  = (∀(γ : ⟦ Γ ⟧C)(s : ⟦ S ⟧τ) → ϕ γ → ξ (γ , s)
                                  → pre ψ e (γ , s))
  × vc (λ (γ , s) → ϕ γ × ξ (γ , s)) ψ e
```

Figure 10. The Agda definition of vc.

To see it in action, we consider the following definitions again:

$$f_0^{\mathsf{A}} = (\lambda x. x + 1) :: \{\nu : \mathbb{N} \mid \nu < 2\} \longrightarrow \{\nu : \mathbb{N} \mid \nu < 4\}$$
$$ex_2^{\mathsf{A}} = (f_0^{\mathsf{A}}\ 1) + 2$$

If we assign ex_2^{A} a postcondition $\lambda\nu. \nu < 6$, then pre computes the weakest precondition to be $1 < 2 \land \delta(f_0^{\mathsf{A}}, 1) + 2 < 6$. It checks the argument 1 against f_0^{A}'s input type, and the whole expression against the given postcondition. The vc function validates that f_0^{A} correctly implements its specification as the type signature sets out.

7.4 Meta-properties of pre and vc

We first state monotonicity lemmas of pre and vc.

Lemma 7.1 (pre is monotone). *For an annotated expression $\Gamma \vdash_A e : T$ in λ^A, if a predicate ψ_1 implies ψ_2, then pre ψ_1 e implies pre ψ_2 e.*

Lemma 7.2 (vc is monotone)**.** *For an annotated expression $\Gamma \vdash_A e : T$ in λ^A, if a predicate ϕ_2 implies ϕ_1, and under the stronger precondition ϕ_2, a postcondition ψ_1 implies ψ_2, then vc ϕ_1 ψ_1 e implies vc ϕ_2 ψ_2 e.*

With the monotonicity lemmas, we can finally prove the soundness and completeness of pre and vc with respect to the typing rules of λ^C.

Theorem 7.3 (Completeness of pre and vc w.r.t. λ^C typing rules)**.** *If $\Gamma; \phi \vdash_C \hat{e} : \{\nu : T \mid \psi\}$, then vc ϕ ψ $\ulcorner \hat{e} \urcorner^C$ and ϕ $\gamma \Rightarrow$ pre ψ $\ulcorner \hat{e} \urcorner^C$ γ for any semantic environment γ that respects Γ.*

Corollary 7.4. *For an expression $\Gamma \vdash_A e : T$ in λ^A, if vc ϕ ψ e and ϕ $\gamma \Rightarrow$ pre ψ e γ for any semantic environment γ that respects Γ, then there is a type derivation $\Gamma; \phi \vdash_C \hat{e} : \{\nu : T \mid \psi\}$ such that $\ulcorner \hat{e} \urcorner^C = e$.*

Theorem 7.5 (Soundness of pre and vc w.r.t. λ^C typing rules)**.** *For an expression $\Gamma \vdash_A e : T$ in λ^A, if vc (pre ψ e) ψ e, then there is a type derivation Γ; pre ψ e $\vdash_C \hat{e} : \{\nu : T \mid \psi\}$ such that $\ulcorner \hat{e} \urcorner^C = e$.*

8 Related Work, Future Work and Conclusion

There is a very long line of prior work on refinement types, e.g. [15, 18, 31, 33, 36], just to name a few. We find the work by Lehmann and Tanter [17] most comparable. They define the language and the logical formulae fully deeply in Coq, and assumes an oracle that can answer the questions about logical entailment. In our formalisation, we interpret the language as shallow Agda terms, and the underlying logic is Agda's type system. Programmers serve as an oracle to construct proof terms. Knowles and Flanagan [14]'s work is also closely related. It develops a decidable type reconstruction algorithm which preserves the typeability of a program. Their type reconstruction is highly influenced by the strongest postcondition predicate transformation.

Admittedly, our attempt in formalising refinement type systems is still in its infancy. We list a few directions for future exploration.

Language features. The languages that we presented in this paper are very preliminary. They do not yet support higher-order functions. Variants of Hoare style logics that deal with higher-order language features [3, 13, 32, 35, 39] will shed light on the extension to higher-order functions. It remains to be seen which techniques are compatible with the way in which the language is embedded and interpreted in Agda. General recursion is also missing from our formalisation. We surmise that recursion can be handled analogously to how a WHILE-loop is dealt with in Hoare logic. Hoare logic style reasoning turns out to be instrumental in languages with side-effects or concurrency. How to extend the unifying paradigm to languages with such features is also an open question. An equivalent question is how to formulate proof systems that support these features in terms of refinement type systems.

Delaying proof obligations. As we have seen in the examples, constructing a typing tree for a program requires the developer to fill the holes with proof terms. The typechecking algorithm with pre and vc collects the proof obligations along the typing tree. This is effectively deferring the proofs to a later stage. It shares the same spirit as the Delay applicative functor by O'Connor [25]. It is yet to be seen how it can be applied in the construction of the typing trees in our formalisation.

Compositionality. We stated in Section 7 that the λ^C language is not yet fully compositional in the sense of [15]. The interpretation function $\mathscr{E}[\![\cdot]\!]^{\mathsf{A}}_{\mathsf{Tm}}$ is used in the definition of pre, and that effectively leaks the behaviour of the program to the reasoning thereof, penetrating the layer of abstraction provided by types. We dealt with it for functions: the implementation details of the function body and those of the argument are hidden from the reasoning process. We would like to further extend the compositional reasoning to other language constructs (via user type annotations) in future work.

Other program logics. Lastly, in our formalisation, we use Hoare logic as the foundation for the typing rules. There are other flavours of program logics, most notably the dual of Hoare logic—Reverse Hoare Logic [5] and Incorrectness Logic [28]. We are intrigued to see if we can mount these logics onto our system, and how it interacts with a functional language that is, say, impure or concurrent.

In this paper, we presented a simple yet novel Agda formalisation of refinement types on a small first-order functional language in the style of Hoare logic. It provides a testbed for studying the formal connections between refinement types and Hoare logic. We believe that our work is a valuable addition to the formal investigation into refinement types, and we hope that this work will foster more research into machine-checked formalisations of refinement type systems, and the connection with other logical systems such as Hoare style logics.

Acknowledgments

We thank Craig McLaughlin for suggesting semantic type soundness, and Carroll Morgan for the helpful discussions about the backward reasoning algorithm outlined in Section 7. The author is supervised by Christine Rizkallah, who provides lots of helpful feedback.

References

[1] Giuseppe Castagna and Alain Frisch. 2005. A Gentle Introduction to Semantic Subtyping. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '05)*. Association for Computing Machinery, New York, NY, USA, 198–199. https://doi.org/10.1145/1069774.1069793

[2] James Chapman. 2009. Type Theory Should Eat Itself. *Electron. Notes Theor. Comput. Sci.* 228 (jan 2009), 21–36. https://doi.org/10.1016/j.entcs.2008.12.114

[3] Nathaniel Charlton. 2011. Hoare Logic for Higher Order Store Using Simple Semantics. In *Logic, Language, Information and Computation*, Lev D. Beklemishev and Ruy de Queiroz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–66.

[4] Nils Anders Danielsson. 2006. A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family. In *Proceedings of the 2006 International Conference on Types for Proofs and Programs (TYPES'06)*. Springer-Verlag, Berlin, Heidelberg, 93–109.

[5] Edsko de Vries and Vasileios Koutavas. 2011. Reverse Hoare Logic. In *Software Engineering and Formal Methods*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 155–171.

[6] Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (Aug. 1975), 453–457. https://doi.org/10.1145/360933.360975

[7] Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (may 2021), 38 pages. https://doi.org/10.1145/3450952

[8] Peter Dybjer. 2000. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic* 65, 2 (2000), 525–549. https://doi.org/10.2307/2586554

[9] Michael Greenberg. 2015. *A refinement type by any other name.* http://www.weaselhat.com/2015/03/16/a-refinement-type-by-any-other-name/

[10] Robert Harper. 2016. *Practical Foundations for Programming Languages* (2nd ed.). Cambridge University Press, USA.

[11] Ranjit Jhala. 2019. Liquid Types vs. Floyd-Hoare Logic. Retrieved 11 May, 2022 from https://ucsd-progsys.github.io/liquidhaskell-blog/2019/10/20/why-types.lhs/

[12] Ranjit Jhala and Niki Vazou. 2021. Refinement Types: A Tutorial. *Foundations and Trends® in Programming Languages* 6, 3–4 (2021), 159–317. https://doi.org/10.1561/2500000032

[13] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[14] Kenneth Knowles and Cormac Flanagan. 2007. Type Reconstruction for General Refinement Types. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*. 505–519. https://doi.org/10.1007/978-3-540-71316-6_34

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

[16] Shinji Kono. 2022. Constructing ZF Set Theory in Agda. Retrieved 9 May, 2022 from https://github.com/shinji-kono/zf-in-agda

[17] Nico Lehmann and Éric Tanter. 2016. Formalizing Simple Refinement Types in Coq. In *The Second International Workshop on Coq for PL (CoqPL'16)*.

[18] Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 775–788. https://doi.org/10.1145/3009837.3009856

[19] Per Martin-Löf. 1984. *Intuitionistic Type Theory.* Bibliopolis.

[20] Conor McBride. 2010. Outrageous but Meaningful Coincidences: Dependent Type-Safe Syntax and Evaluation. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming (WGP '10)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1863495.1863497

[21] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375.

[22] Tobias Nipkow and Gerwin Klein. 2014. *Concrete Semantics with Isabelle/HOL*. Springer. https://doi.org/10.1007/978-3-319-10542-0

[23] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory.* PhD Thesis. Department of Computer Science and Engineering, Göteborg, Sweden.

[24] Ulf Norell. 2009. Dependently typed programming in Agda. In *Proceedings of the 4th international workshop on Types in language design and implementation (TLDI '09)*. ACM, New York, NY, USA, 1–2. https://doi.org/10.1145/1481861.1481862

[25] Liam O'Connor. 2019. Deferring the Details and Deriving Programs. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe 2019)*. Association for Computing Machinery, New York, NY, USA, 27–39. https://doi.org/10.1145/3331554.3342605

[26] Liam O'Connor. 2019. *Type Systems for Systems Types.* Ph. D. Dissertation. UNSW, Sydney, Australia. http://handle.unsw.edu.au/1959.4/64238

[27] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. 2016. Refinement Through Restraint: Bringing Down the Cost of Verification. In *International Conference on Functional Programming*. Nara, Japan. https://trustworthy.systems/publications/nicta_full_text/9425.pdf

[28] Peter W. O'Hearn. 2019. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL (dec 2019), 32 pages. https://doi.org/10.1145/3371078

[29] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. *Exploring New Frontiers of Theoretical Informatics: IFIP 18th World Computer Congress TC1 3rd International Conference on Theoretical Computer Science (TCS2004) 22–27 August 2004 Toulouse, France.* Springer US, Boston, MA, Chapter Dynamic Typing with Dependent Types, 437–450. https://doi.org/10.1007/1-4020-8141-3_34

[30] Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. Cogent: uniqueness types and certifying compilation. *Journal of Functional Programming* 31 (2021). https://doi.org/10.1017/S095679682100023X

[31] Zvonimir Pavlinovic, Yusen Su, and Thomas Wies. 2021. Data Flow Refinement Type Inference. *Proc. ACM Program. Lang.* 5, POPL (jan 2021), 31 pages. https://doi.org/10.1145/3434300

[32] Yann Régis-Gianas and François Pottier. 2008. A Hoare Logic for Call-by-Value Functional Programs. In *Mathematics of Program Construction*, Philippe Audebaud and Christine Paulin-Mohring (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 305–335.

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

[34] Amr Sabry and Matthias Felleisen. 1992. Reasoning About Programs in Continuation-passing Style. *SIGPLAN Lisp Pointers* V, 1 (Jan. 1992), 288–298.

[35] Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. 2009. Nested Hoare Triples and Frame Rules for Higher-Order Store. In *Proceedings of the 23rd CSL International Conference and 18th EACSL Annual Conference on Computer Science Logic (CSL'09/EACSL'09)*. Springer-Verlag, Berlin, Heidelberg, 440–454.

[36] Niki Vazou. 2016. *Liquid Haskell: Haskell as a Theorem Prover.* Ph. D. Dissertation. University of California, San Diego, USA.

[37] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. LiquidHaskell: Experience with Refinement Types in the Real World. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. ACM, New York, NY, USA, 39–51. https://doi.org/10.1145/2633357.2633366

[38] A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (nov 1994), 38–94. https://doi.org/10.1006/inco.1994.1093

[39] Nobuko Yoshida, Kohei Honda, and Martin Berger. 2007. Local State in Hoare Logic for Imperative Higher-Order Functions. In *Proc. Fossacs (LNCS, Vol. 4423)*. Springer, 361–377.