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 small calculus. In our formalisation, we use shallow Agda terms as the denotation for the object language and also use Agda 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.

Keywords: Refinement types, Hoare Logic, Agda

1 Introduction

Refinement types is a lightweight yet expressive tool for specifying and reasoning about programs. The programmers annotate their programs with types, which can include predicates to further restrict the inhabitants of that type. For instance, $\{\nu : \mathbb{N} \mid \nu > 0\}$ is a type for all 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 are complicated in several ways. Typically, a refinement type system supports *dependent functions*, which is similar to those in a dependent type system [17]. Dependent functions allow the refinement predicate to refer to the value of the function's argument. 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 complication in refinement type systems is solving the logical entailment which determines the subtyping relation between two types. Usually, some tactics based on syntactic or semantic rewriting will be involved to carefully transform the entailment into a certain form, to facilitate the SMT-solver to automatically discharge the proof obligations. For the SMT-solving to be decidable, language designers typically need to restrict the logic of the refinement predicates. For instance, in Liquid Haskell [33], the quantifier-free logic of equality, uninterpreted functions and linear arithmetic (QF-EUFLIA) is used.

Due to these complications, the development on refinement types remains largely informal[1] (with the exception of the work by Lehmann and Tanter [15], to the best of our knowledge), and ad hoc to some degrees. 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.

Refinement types and Hoare logic have some deep connections, which has long been recognised. For example, the work by Jhala and Vazou [11] makes references to Hoare logic throughout their development. It summarises the relationship between the two systems as:

> The development [...] shows that refinement types can be viewed as a generalization of Floyd-Hoare style program logics. Such logics typically have monolithic assertions that describe the entire state of the machine at a given program point. Types allow us to decompose those assertions into more fine-grained refinements on the values of individual terms. Similarly, pre- and post-conditions correspond directly to input- and output-types for functions.

In a blog post [10], Jhala further explains why Liquid Types are different (and in some aspects, superior to) Hoare logic, with the punchline "types decompose quantified assertions into quantifier-free refinements". With a refinement type system, in which logical formulas can be put in type positions, it eliminates the use of universal quantifiers in them, rendering the verification conditions more decidable in SMT solvers. It also, due to parametricity of types, provides a way of relating different objects.

The formal connections between refinement types and Hoare logic deserve more systematic studies. In this paper, we present a unifying paradigm – a Hoare-triple style formalisation of a refinement type system on a small purely functional language based on λ-calculus. Formalising refinement types in the Hoare logic style not only allows us to study the connections between these two systems, it also makes the

TyDe'22, 11 September, 2022, Ljubljana, Slovenia

[1]Informal in the sense of lacking machine-checked formalisations.

formalisation easier by avoiding the aforementioned complications in refinement type systems. The formalisation is done in Agda [21, 22], a dependently typed theorem prover. In our formalisation, we use shallow Agda terms as the denotation for the object language and also use Agda's type system as the underlying logic for the type refinement.

In a nutshell, we formulate the typing rules of the refinement type system as $\Gamma\{\phi\} \vdash e : T\{\psi\}$. When reading it as a regular typing rule, the typing context is split into two parts: Γ is a list of term variables associated to base types, and ϕ contains all the refinement predicates about these variables in the context. e is the expression being typechecked, and T and ψ form the refinement type that e is checked against. 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.

When we make the analogy between refinement type systems and Hoare logic, another analogy naturally arises. The typechecking of a refinement type system has some connection with the weakest precondition in Hoare logic. In fact, the idea of using weakest precondition for refinement typechecking is not new. Knowles and Flanagan [12] has proposed as future work to propagate information backwards, calculating the weakest precondition as an avenue to refinement type reconstruction. In this paper, we explore how to use backwards reasoning for typechecking, with our machine-checked formalisation in Agda.

Specifically, this paper makes the following technical contributions:

- We formalise a refinement type system (Section 3 and Section 4) à la Hoare logic, and prove meta-properties of the static semantics of the language (Section 5).
- We define a naïve weakest precondition function wp in lieu of a typechecking algorithm and prove meta-properties about it (Section 6).
- We refine the formalisation above and present a variant of the refinement type system which preserves the contracts imposed by functions (i.e. λ-abstractions), which requires a more sophisticated weakest precondition function pre and a verification condition generator vc. We prove the soundness and completeness of pre and vc with respect to the typing rules (Section 7).

All the formalisation is developed in and checked by Agda (version 2.6.2.1), and the semantics of the object language is interpreted as Agda terms. In fact, the main body of this paper is generated from a literate Agda file, which contains all the formal development. The source file of this paper can be obtained at 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[2], a refinement type is defined as a record of a value, and an irrelevant proof of a property P over the value:

```
record Refinement′ {a p} (A : Set a) (P : A → Set p)
                                        : Set (a ⊔ p) where
  constructor _,_
  field value : A
        proof : Irrelevant (P value)
```

One tedium in defining and working with such an encoding is that the object language also features term-dependent types. Encoding a dependently-typed language in another dependently-typed language often involves using inductive-recursive techniques [8]. The dependent object language features telescopic contexts in the typing rules. As such, it poses extra challenges in manipulating the contexts, and in performing type inference in general, as the dependency induces specific topological orders in solving type constraints.

Realising the connection between refinement types and Hoare logic can be a rescue. When assigning a refinement type to a function (we assume that all functions only take one argument), the refinement predicate on the argument asserts the properties of the input, and the predicate on the result type needs to be satisfied by the output. This mimics the structure of a Hoare-triple, in a way that the former corresponds to the precondition and the latter to the postcondition. A slightly less obvious correlation is that, in a typing judgement $\Gamma \vdash e : \tau$, the refinement predicates in the typing context Γ correlate to the precondition, and the refinement predicate in τ corresponds to the postcondition of "executing" the expression e.

With that observation, in a typing judgement $\overrightarrow{x_i : \tau_i} \vdash e : \{\nu : B \mid \psi(\overline{x_i}, \nu)\}$[3] for refinement types, we can pull out the refinement predicates in $\vec{\tau_i}$, forming a precondition over the binders in the context, which is analogous to the precondition on the program state in Hoare logic for imperative languages. We aggregate all the refinement predicates and take their conjunction, which is a predicate of type $\phi : \overline{\mathrm{base}(\tau_i)} \rightarrow \mathrm{Set}$, where base is a function that extracts the base type of a refinement type, and Set is the type of propositions. Similarly, ψ can be deemed as the postcondition after e has been executed. In a purely functional language, it amounts to a predicate over the value of e and the variables $\overline{x_i}$ in the context.

The Hoare-triple view of refinement types has many benefits. Firstly, it separates the typechecking of the base types and the that of the refinement predicates, which is a common practice in refinement typed languages (e.g. [12, 29, 30]). In our system, the "type system" is simply-typed, whose type

[2] Our Agda development uses the following commit of agda-stdlib: https://github.com/agda/agda-stdlib/blob/95270b78d/src/Data/Refinement.agda

[3] We use an $\overrightarrow{\text{overhead arrow}}$ to denote an ordered vector, and an $\overline{\text{overhead line}}$ for an unordered list.

inference is well-established. With the base types out of the way, it allows us to focus on the refinements. 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 the predicates ϕ and ψ above, the variables x_i no longer need to maintain any particular order.

In our formalisation, we factor out the automation of a decidable type inference algorithm with an SMT-solver, which is often desirable in refinement typed languages. In the small calculus that we study, we require all functions (λ-abstractions) to be annotated with types and they are the only places that type annotations are needed. We only perform typechecking, without elaborating the entire syntax tree. 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. The proof of the verification conditions are left to the users, who serve as an oracle for solving all logic puzzles.

3 The Base Language λ^B

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

The syntax of the term language is largely canonical with one peculiarity, which is how functions are defined. For reasons that we will see later when we come to the formalisation of refinement types, we define function types and function terms (λ-abstractions) as their own syntactic groups. The main purpose is that we can later define inductive rules and functions on types and expressions in a syntactic manner. As we will see later, the Agda formalisation of the language is interpreted in a tagless manner [1]. This syntactic distinction injects a little bit of taggedness to it, allowing us to dispatch depending on the syntax of the objects more easily. Conceptually, this distinction is not always relevant, especially in the pen-and-paper formalisation. Therefore whenever possible, we only define a single inductive definition or function on paper, and it maps to two definitions or functions in the Agda formalisation.

base types	B, S, T	::=	$\mathbb{1} \mid \mathbb{2} \mid \mathbb{N} \mid S \times T$
func. types		∋	$S \longrightarrow T$
expressions	e	::=	$x \mid () \mid \mathsf{true} \mid \mathsf{false}$
		|	$\mathsf{ze} \mid \mathsf{su}\ e$
		|	$(e, e) \mid \pi_1\ e \mid \pi_2\ e \mid f\ e$
		|	$\mathbf{if}\ c\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$
		|	$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$
		|	$e_1 + e_2 \mid e_1 - e_2 \mid e_1 < e_2$
functions	f	::=	$\lambda x.e$
contexts	Γ	::=	$\cdot \mid \Gamma, x : S$

Figure 1. Syntax of the language λ^B

Since its typing rule is very standard, we directly show how we encode it in Agda and use it as a tutorial on how we construct the language in Agda. We use an encoding directly derived from McBride's Kipling language [18], which allows us to index the syntax of the object language with its type in Agda. It effectively lets Agda to perform typechecking up to simple types as we construct the syntax of the term language.

We first introduce some auxiliaries before diving into the Agda definition of λ^B. For more details of the general set up, we recommend readers to consult McBride [18]'s work.

McBride [18] uses inductive-recursive definitions [8] for the dependent types in his object language, which is a pretty standard technique used in embedding dependently type languages(e.g. [3, 4]). In our base language (and also later with refinement types), however, since term-dependency in types has been eliminated, 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.

We define a universe **U** of deep base types, and an interpretation function $\mathscr{E}[\![\_]\!]_{\mathsf{Ty}}$ which maps the syntax to Agda types. We do not include a code for function types; they will be handled by the typing rules separately.

```
                          ⟦_⟧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
```

Following the work on semantic typing [19, 34], we define what it means for a denotational value to possess a type.

Definition 3.1. 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.

Next, we define the typing context for the simply-typed language λ^B, and the denotation of the context in terms of a nested tuple of Agda values. The denotation of the typing context gives us a *semantic environment* γ, mapping variables to denotational values in Agda. The semantic environment γ obtained from the denotation function *respects* the typing context in the sense that for all $x \in \mathrm{dom}(\Gamma)$, $\vDash \gamma(x) : \Gamma(x)$.

The typing context and its denotation function $\mathscr{E}[\![\_]\!]_{\mathsf{Cx}}$ are defined in Agda as follows:

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

The context is nameless and uses de Bruijn indices for context operations, with the rightmost (also outermost) element bound most closely. Unlike Kipling [18], the direction to which the context grows is largely irrelevant, since the context is not telescopic. We define the syntax for context lookup and its interpretation in Agda:

```
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
```

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 and ˆ and ˇ are synonyms for the currying and uncurrying functions respectively.

The syntax of the language is defined in Agda as follows:

```
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
  ADD   : Γ ⊢ `ℕ´ → Γ ⊢ `ℕ´ → Γ ⊢ `ℕ´
  MINUS : Γ ⊢ `ℕ´ → Γ ⊢ `ℕ´ → Γ ⊢ `ℕ´
  LT    : Γ ⊢ `ℕ´ → Γ ⊢ `ℕ´ → Γ ⊢ `2´

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

We index the type of the deep terms with the typing context and the type of the term. Therefore the terms respect the typing rules by construction. The syntax for the term language (and also its typing rules) is very standard. We only mention that in a function application `APP`, only function expressions can be applied to an argument. `FUN` has the same type as a normal first-class λ-abstraction does. It can be constructed under any context Γ, and it supports closures.

In our language, arithmetic operations are defined as primitive language constructs. We deviate from McBride [18]'s generic recursion principle `REC` for natural numbers, as it is very cumbersome to define other language constructs in terms of `REC`, and also because our types in the object language are not dependent. We discuss the implications of adding general recursion to the refinement typed language at the end of the paper.

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

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

it can be done in Agda as

```
f₀ : ∀{Γ} → Γ ⊢ `ℕ´ ⟶ `ℕ´
f₀ = FUN (ADD ONE (VAR top))
```

where `ONE` is defined to be `SU ZE`. Note that the function's type is parametric in the context Γ.

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

```
⟦_⟧⊢ : ∀{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 ⟧⊢ = ˆ ⟦ e₂ ⟧⊢ ˢ ⟦ e₁ ⟧⊢
⟦ PRD e₁ e₂ ⟧⊢ = < ⟦ e₁ ⟧⊢ , ⟦ e₂ ⟧⊢ >
⟦ FST e ⟧⊢ = proj₁ ∘ ⟦ e ⟧⊢
⟦ SND e ⟧⊢ = proj₂ ∘ ⟦ e ⟧⊢
⟦ APP f e ⟧⊢ = ⟦ f ⟧⊢⃗ ˢ ⟦ e ⟧⊢
⟦ ADD e₁ e₂ ⟧⊢ = ᵏ _+_ ˢ ⟦ e₁ ⟧⊢ ˢ ⟦ e₂ ⟧⊢
⟦ MINUS e₁ e₂ ⟧⊢ = ᵏ _-_ ˢ ⟦ e₁ ⟧⊢ ˢ ⟦ e₂ ⟧⊢
⟦ LT e₁ e₂ ⟧⊢ = ᵏ _<ᵇ_ ˢ ⟦ e₁ ⟧⊢ ˢ ⟦ e₂ ⟧⊢

⟦ FUN e ⟧⊢⃗ = ˆ ⟦ e ⟧⊢
```

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 2. Syntax for the language λ^R

The denotation of the above `f₀` function under any 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 → suc x`, independent of the environment γ.

4 Refinement Typed Language λ^R

Adding refinement predicates to the λ^B typing judgement results in the typing judgement for the refinement typed language λ^R. We first present the syntax of the language in Figure 2, in addition to Figure 1.[4] The upcast operator for non-function expressions is used to make any subtyping explicitly in the typing tree.

Apart from the way we organise function arrows in refinement types as mentioned above, another difference is how typing contexts are defined. Instead of a vector of $x_i : \tau_i$ entries, we separate the predicates and the base types. The context therefore becomes $\Gamma;\phi$, where $\Gamma ::= \overline{x_i : \text{base}(\tau_i)}$ and ϕ is the conjunction of the predicates gathered from τ_is. These two formulations are informally equivalent. Typically, in a typing context of the form $\overrightarrow{x_i : \tau_i}$, where each τ_i is a refinement type, additional (path sensitive) constraints can be added to the context in $\Gamma \vdash e : \tau$ by introducing a fresh variable of an arbitrary base type, such as $y : \{\nu : \mathbb{1} \mid \phi\}$, where y is not free in e and ν is not free in ϕ. In our formulation, this is made even easier; additional conjuncts can be added to predicate ϕ directly.

We define predicates as a shallow function in Agda, from a vector of variables that are in the domain of a semantic environment to the Agda `Set`. We also define a substitution function in Agda which allows us to substitute the top-most variable in a predicate by an expression.

```
-- substitution
```

[4]As a remark on the notation, when we talk about the dependent function types in our language, we use a slightly longer function arrow $\longrightarrow$ as a reminder that it is not a first-class type constructor. The typesetting is only subtly different from the normal function arrow $\rightarrow$ and in fact its semantics is similar to the normal function arrow. So in reading and understanding the rules, they can be considered identical.

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

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

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

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

Figure 3. Well-formedness rules for contexts and types

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

In Figure 3, 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 4. 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 variable x. In many other calculi (e.g. [11, 13, 27]), a selfification rule is used for variables:

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

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

Constants. For value constants ((), true, false, ze) and function constants (+, −, <, (_, _), π_1, π_2, su), we always infer the exact types for the results. As with the VAR$^{\text{R}}$ rule, we do not keep the refinement predicates in the premises in the refinement type in the conclusion. Again, no information is lost during this process, as they can be recovered later from the context when needed. In practice, a minor drawback is that, some of the proofs will need to be reconstructed. But luckily, most of the time we can simply assign intermediate expressions (i.e. those in the premises in the rules) trivial refinement types.

Let-bindings. In the LET$^{\text{R}}$ rule, ϕ, as usual, is the predicate over the context Γ. ξ is a predicate on the entries in the context Γ and the **let**-binder x. The former can be inferred by the fact that ϕ appears in $\Gamma;\phi$, and the latter by ξ being the refinement predicate of e_1. What is worth noting is that we have to explicitly state that $\{\nu : T \mid \psi\}$ is wellformed under the context Γ instead of the extended $\Gamma, x : S$. This way we ensure that ψ does not mention the locally bound variable x

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}\hat{\Gamma} \vdash_R \hat{e}_1 : \{\nu : \mathbb{N} \mid \psi_1\} \\ \hat{\Gamma} \vdash_R \hat{e}_2 : \{\nu : \mathbb{N} \mid \psi_2\} \qquad \oplus \in \{+, -, <\}\end{array}}{\hat{\Gamma} \vdash \hat{e}_1 \oplus \hat{e}_2 : \{\nu : \mathbb{N} \mid \nu = \hat{e}_1 \oplus \hat{e}_2\}}\ \mathsf{ADD}^{\mathsf{R}}/\mathsf{MINUS}^{\mathsf{R}}/\mathsf{LT}^{\mathsf{R}}$$

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

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

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

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

Figure 4. Static semantics of λ^R

and leaks the implementation details of e_1 into the resulting type.

Function applications. The typing rule for function application is pretty standard. In the work of Knowles and Flanagan [13], a compositional version of this rule is used instead. To summarise the key idea, consider the following rule. A typical function application rule (including ours) has the following form:

$$\frac{\Gamma \vdash f : (x : \tau_1) \to \tau_2 \qquad \Gamma \vdash e : \tau_1' \qquad \Gamma \vdash \tau_1' \sqsubseteq \tau_1}{\Gamma \vdash f\ e : \tau_2[e/x]} \tag{1}$$

In the resulting type, the term e is substituted for x. This would get around the type abstraction on e, exposing the implementation details of the argument to the resulting refinement type $\tau_2[e/x]$, and also rendering the type $\tau_2[e/x]$ arbitrarily large due to the presence of e. To rectify this problem, Knowles and Flanagan [13] propose the result type to be existential: $\exists x : \tau_1'. \tau_2$. Which application rule to include largely depends on the language design, and appears to be orthogonal to the focus of this paper. We use the traditional one here and the compositional one later in this paper as a comparison.

Jhala and Vazou [11] sticks to the regular function application rule, but with some extra restrictions. They require the function argument to be in A-normal form (ANF) [31], 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-theoretic proofs.

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

$$\frac{\Gamma, x : S \vDash \phi \Rightarrow \phi'}{\Gamma \vdash \{\nu : S \mid \phi\} \sqsubseteq \{\nu : S \mid \phi'\}}\ \mathsf{Sub}$$

$$\frac{\Gamma \vdash S_2 \sqsubseteq S_1 \qquad \Gamma, x : S_2 \vdash T_1 \sqsubseteq T_2}{\Gamma \vdash x : S_1 \to T_1 \sqsubseteq x : S_2 \to T_2}\ \mathsf{Sub\text{-}Fun}$$

The subtyping rules are (at least partly) syntactic. In our language, since we do not yet support higher-order functions, the subtyping rule for functions is not needed. It can be achieved by promoting the argument and promoting the result of a function application. The syntactic distinction in our formalisation allows us to define subtyping exclusively for non-function typed expressions. If we included function types in the group of base types, and allowed refinement predicates over function spaces, the syntax would be quite a bit simpler than it is now. However it would then require a full semantic subtyping relation that also works on the function space. This has been shown to be possible, for example interpreting the types in a set-theoretic fashion as in Castagna and Frisch [2]'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. [14] is an attempt to define Zermelo–Fraenkel set theory **ZF** in Agda).

In our system, we directly define the subtyping-style rules ($\mathsf{SUB}^{\mathsf{R}}$, $\mathsf{WEAK}^{\mathsf{R}}$) in terms of logical entailments: $\phi \vDash \psi \Rightarrow$

$\psi' \overset{def}{=} \forall \gamma\ x. \phi\ \gamma \wedge \psi\ (\gamma, x) \rightarrow \psi'\ (\gamma, x)$. This is made possible because the predicates are separate from the types in our formalisation.

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 (1) 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 $\hat{\Gamma} \vdash \hat{e} :: \tau$ to mean $\hat{\Gamma} \vdash \hat{e} :: \tau : \tau$, 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, on the contrary, the precondition P is not involved in the implication. The 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 on the program states, which typically include mappings from variables to values. A variable assignment statement or reference update operation will change the state. Therefore after the execution of 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 on the typing context Γ. The typing judgement does not invalidate the predicate ϕ. Moreover, in practice, the user of a refinement type system will only state predicates of the term being typed, without including information about other variables that are not directly relevant. Therefore it is technically possible not to require ϕ in the implication, but it renders the system really unwieldy to use in practice.

As for the weakening rule, contrary to the more canonical structural weakening rule [15]:

$$\frac{\vdash \Gamma_1, \Gamma_2, \Gamma_3 \qquad \Gamma_1, \Gamma_3 \vdash e : T}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash e : T}$$

our weakening rule only changes the predicates in the context; it does not touch the simply-typed portion of the context and does not allow for adding new binders to the context. It compares favourably to those with a more syntactic refinement-typing context. When the context is defined as $\overrightarrow{x_i : \{\nu : B_i \mid \phi_i\}}$, 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 construct a λ-abstraction under any context Γ and closure is supported. 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 is not very formal in every aspect. One 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 program objects 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 program terms into logic using the already-defined interpretation function ⟦_⟧⊢, with which we interpret the object language as Agda terms.

Definition 4.1 (Erasure)**.** The erasure function $\ulcorner\_\urcorner^R$ removes all refinement type information from an 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 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 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 e can be assigned a refinement type $\{\nu : T \mid \psi(\overline{x_i}, \nu)\}$, where x_i are the entries in the context Γ. For functions, we have a data type `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 data type and the erasure function $\ulcorner\_\urcorner^R$ are inductive-recursively defined.

The context weakening rule WEAK^R in Figure 4 is in fact admissible in our system, therefore it is not included as a primitive construct in the formal definition of the language.

Lemma 4.2 (Weakening is admissible)**.** *For any typing tree $\Gamma; \phi \vdash \hat{e} : \tau$, if $\phi' \Rightarrow \phi$ under the semantic environement γ that respects Γ, then there exists a typing tree with the stronger context $\Gamma; \phi'$, such that $\Gamma; \phi' \vdash \hat{e}' : \tau$ and $\ulcorner \hat{e} \urcorner^R = \ulcorner \hat{e}' \urcorner^R$.*

Proof. By induction on $\hat{e}$. □

Continuing on the previously defined `f₀` function, if we want to lift it to a function definition in λ^R, we will need to insert some explicit upcast nodes:

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

In Agda, it is defined as follows:

```
f₀ᴿ : `E´ { ᵏ ⊤ }⊢ `ℕ´ { (_< 2) ∘ proj₂ }⟶
                 `ℕ´ { (_< 4) ∘ proj₂ }
f₀ᴿ = FUNᴿ (SUBᴿ (ADDᴿ (VARᴿ top) ONEᴿ)
             _λ (_ , s) t s<2 t≡s+1 →
               s<2⇒t≡s+1⇒s+1<4 s<2 t≡s+1)
```

The upcast node $\mathsf{SUB}^{\mathsf{R}}$ needs to be accompanied by an evidence (i.e. an Agda proof object) to justify the semantic entailment $x < 2 \vDash \nu = x + 1 \Rightarrow \nu < 4$.

To demonstrate the function application of $\mathsf{f}_0^{\mathsf{R}}$, we define the following expression:

$$ex_0^{\mathsf{R}} : \{\nu : \mathbb{N} \mid \nu < 5\}$$
$$ex_0^{\mathsf{R}} = f_0^{\mathsf{R}}\ (1 :: \{\nu : \mathbb{N} \mid \nu < 2\}) :: \{\nu : \mathbb{N} \mid \nu < 5\}$$

The inner upcast is for promoting the argument 1, which is inferred the exact type $\{\nu : \mathbb{N} \mid \nu = 1\}$, to match the contract laid out by the function f_0^{R}'s type signature. The outer upcast is to promote from the designated f_0^{R}'s result type $\{\nu : \mathbb{N} \mid \nu < 4\}$ 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≡1⇒s<2 s≡1))
           _λ _ _ _ t<4 → t<4⇒t<5 t<4
```

5 Meta-Properties of λ^R

Instead of proving the textbook type soundness theorems (progress and preservation) [9, 34] that rest upon subject reduction, we instead get for free the type soundness theorem à la Milner [19] that is based on denotational semantics.

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

The shallow denotation of the simply-typed term language automatically guarantees that the Agda denotation of a term possesses the type to which the type system of λ^B assigns the term. The theorem above, however, does not state the type soundness property of refinement types. Next we introduce the formalisation of the refinement soundness theorem. 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.

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

We can give meanings to refinement types in the following way:

Definition 5.3. A value v posesses a refinement type $\{\nu : T \mid \psi\}$, written $\vDash v : \{\nu : T \mid \psi\}$, if $\vDash v : T$ and $\varnothing \vDash \psi[v/\nu]$.

With this interpretation of refinment types, we can proceed with the (semantic) type soundness theorem with respect to refinement types:

Theorem 5.4 (Refinement soundness). *If $\Gamma; \phi \vdash \hat{e} : \{\nu : T \mid \psi\}$, then $\phi\ \gamma \vDash \psi[\mathscr{E}[\![\ulcorner \hat{e} \urcorner^R]\!]_{Tm}\gamma/\nu]$, where γ respects the typing context Γ and satisfies ϕ.*

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

Theorem 5.5 (Refinement completeness). *If for all expression e, such that $\Gamma \vdash e : T$ and for all semantic context γ that respects Γ and satisfies ϕ, $\phi\ \gamma \vDash \psi[\mathscr{E}[\![e]\!]_{Tm}\gamma/\nu]$ is true, then there must exists $\hat{e}$ and $\Gamma; \phi \vdash \hat{e} : \{\nu : T \mid \psi\}$, such that $\ulcorner \hat{e} \urcorner^R = e$.*

The proofs of Theorem 5.4 and Theorem 5.5 are both easy inductions on the typing tree. 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 $\mathsf{SUB}^{\mathsf{R}}$ and $\mathsf{WEAK}^{\mathsf{R}}$ rules.

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

Corollary 5.6. *Refinement soundness holds for closed terms.*

Proof. Instantiate Γ with $\varnothing$ in Theorem 5.4. ☐

Corollary 5.7. *For refinement typing judgements, the predicate ϕ on the context is an invariant, namely, $\Gamma; \phi \vdash_R \hat{e} : \{\nu : T \mid \lambda\nu. \phi\}$.*

Proof. By Theorem 5.5. ☐

Corollary 5.8 (Consistency). *It is impossible to assign a void refinement type to an expression $\Gamma; \phi \vdash_R \hat{e} : \{\nu : T \mid \lambda\nu. \bot\}$, if $\varnothing \vDash \phi\ \gamma$ for any semantic environment γ that respects Γ.*

Proof. By Theorem 5.4. ☐

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]. In an imperative language, when a variable x gets assigned, the Hoare triple is $\{Q[e/x]\}\ x := e\ \{Q\}$, which means that the precondition can be acquired by simply substituting the variable x in the postcondition Q by the expression to which the variable is assigned. This precondition is in fact also the weakest precondition. In our formulation

$\Gamma\{\phi\} \vdash e : T\{\lambda\nu.\psi\}$, the postcondition ψ is a predicate over the result of the expression e and the context, with ν binding the value of e. In a purely functional setting, since nothing changes before and after the typing of e, anything true of e must have been true in the context. Therefore if we substitute e for ν in ψ, it becomes a predicate over the binders in the typing context, of which e is composed.

Definition 6.1. For any expression $\Gamma \vdash e : T$ in the language λ^B, if the postcondition over e is $\lambda\nu.\psi$, then the weakest precondition is defined as $\mathsf{wp}\ \psi\ e = \psi[e/\nu]$.

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 5.4 and Theorem 5.5) respectively.

Theorem 6.2 (Completeness of wp w.r.t. λ^R typing)**.** *If* $\Gamma; \phi \vdash_R \hat{e} : \{\nu : T \mid \psi\}$, *then* $\phi\ \gamma \Rightarrow wp\ \psi \ulcorner e \urcorner^R\ \gamma$ *for any semantic environment* γ *that respects* Γ.

Theorem 6.3 (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 exists 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, it can infer the weakest precondition for it to be 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 that if an expression is typeable by inferring a weakest context, but it does not elaborate the typing tree, annotating each sub-expression with a type, nor does it automatically construct the proof terms. Despite the limitation, this method can still be applied to program verification tasks in which the exact typing tree does not need to be constructed, or when the construction of the proof terms do not need to be automatic. For instance, we intent to augment the Cogent language [24, 25, 28], a purely functional language for easing the formal verification of systems code, with refinement types. In Cogent's verification framework, a fully elaborated typing tree is not necessary, and the functional correctness of a system is manually specified and 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.

7 Function Contracts With λ^C

As we have seen in the last few sections, such a backwards typechecking algorithm is very easy to define and works uniformly on all terms. The language λ^R, however, has a very unfortunate drawback – it does not respect the type signatures given to functions. For example, the wp algorithm does not check the argument to a function has the right type, nor does it check that a function's definition satisfies its type signature. After all, the type signatures are erased in the program that wp operates on, and syntax tree is lost as the object language expressions get reflected into the logic as shallow Agda terms. As a concrete illustration, when we apply wp to the ex_0^R program above, it only returns a verification condition $2 < 5$ for the whole program, whereas no check on the argument or on the function f_0's definition is being conducted.

```
wp-ex₀ : _
wp-ex₀ = wp ((_< 5) ∘ proj₂) ex₀ -- 2 < 5
```

To rectify the problem, we define a variant of the language λ^C, which is compositional in the sense that the function contracts are respected.[5] It is worth mentioning that the language is not yet compositional in the sense of [13], 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. We only makes two changes in the typing rules for λ^C:

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

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

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

As suggested by Knowles and Flanagan [13]'s work, 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^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:

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

$$\text{inc} : (x : \mathbb{N}) \to \{\nu : \mathbb{N} \mid \nu = x + 1\}$$
$$\text{inc} = \lambda x.\ \text{su}\ x$$

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. This turns out to be important in making the typechecking algorithm deterministic – no guess work is needed for ξ. The LET$^{\mathsf{C}}$ will work equally well in the λ^R language, but the typing rules (or, program logic) for λ^C need to be more carefully chosen, as the implementation of functions are abstracted away from the reasoning process.

Since the definition of the λ^C in the Agda formalisation is indexed by the typing rules, when the typing rules change, the deep syntax of the language has to be defined again. We also define the erasure function $\ulcorner\_\urcorner^{\mathsf{C}}$ on λ^C similar to the $\ulcorner\_\urcorner^{\mathsf{R}}$ function presented before. The λ^C is interpreted the same way as λ^R, by taking the Agda denotation of the erased terms.

7.2 Annotated untyped language λ^A

To perform refinement typechecking on λ^C, we define a variant of a base language λ^A, which is identical to the simply-typed base language λ^B, except that the functions are associated with type annotations for its input and output types. We denote function expressions in λ^A as $f :: (x : \xi) \to \psi$, instead of an untyped bare f.

In order to interpret the annotated language λ^A, following our tradition we define an erasure function $\ulcorner\_\urcorner^{\mathsf{A}}$ to the base language λ^B, so that the Agda denotation of λ^A can be obtained by using λ^B as an intermediary. To establish the connection between λ^C and λ^A, another partial erasure function $\ulcorner\_\urcorner^{\mathsf{B}}$ is defined, to take a λ^C term to the corresponding λ^A term.[6] It preserves the function's type annotation in λ^C, so that we know that when a λ^A term is typed, the functions are typed as prescribed. With the three erasure functions, we prove that they form a commuting diagram:

Lemma 7.1. *For all expression $\hat{e}$ in λ^C, $\ulcorner\ulcorner\hat{e}\urcorner^B\urcorner^A = \ulcorner e\urcorner^C$.*

Proof. By induction on $\hat{e}$. □

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 did 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 [20, §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 [20]'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. pre immediately returns the pre-condition of a function, which is the refinement predicate on the argument type of the function. vc additionally checks that the provided function signatures make sense. 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 inferred from the program context. Additionally, vc needs to recurse down the syntax tree and gather verification conditions from sub-expressions, and, in particular, descent into function definitions to check that they meet the given type signatures. The definitions of the pre and the vc functions are shown below:[7]

```
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)
```

[6]The superscript B in $\ulcorner\_\urcorner^{\mathsf{B}}$ is because, B is in between A ($\ulcorner\_\urcorner^{\mathsf{A}}$)and C ($\ulcorner\_\urcorner^{\mathsf{C}}$).

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

```
pre ψ (SUᴬ e) = pre (ᵏ ⊤) e ∩ ψ [ ⌜ SUᴬ e ⌝ᴬ ]s
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 ⌝ᴬ ]s
pre ψ (FSTᴬ e) = pre (ᵏ ⊤) e ∩ ψ [ ⌜ FSTᴬ e ⌝ᴬ ]s
pre ψ (SNDᴬ e) = pre (ᵏ ⊤) e ∩ ψ [ ⌜ SNDᴬ e ⌝ᴬ ]s
pre _ (APPᴬ {ξ = ξ}{ψ = ψ} f e) = pre ξ e
pre ψ (ADDᴬ e₁ e₂)
  = pre (ᵏ ⊤) e₁ ∩ pre (ᵏ ⊤) e₂ ∩ ψ [ ⌜ ADDᴬ e₁ e₂ ⌝ᴬ ]s
pre ψ (MINUSᴬ e₁ e₂)
  = pre (ᵏ ⊤) e₁ ∩ pre (ᵏ ⊤) e₂ ∩ ψ [ ⌜ MINUSᴬ e₁ e₂ ⌝ᴬ ]s
pre ψ (LTᴬ e₁ e₂)
  = pre (ᵏ ⊤) e₁ ∩ pre (ᵏ ⊤) e₂ ∩ ψ [ ⌜ LTᴬ e₁ e₂ ⌝ᴬ ]s
pre ψ e = ψ [ ⌜ e ⌝ᴬ ]s -- It's just subst for the rest

pre→ {ξ = ξ}{ψ = ψ} (FUNᴬ e) = ξ ∩ pre ψ e

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

vc ϕ ψ (VARᴬ x) = ⊤
vc ϕ ψ UNITᴬ = ⊤
vc ϕ ψ TTᴬ = ⊤
vc ϕ ψ FFᴬ = ⊤
vc ϕ ψ ZEᴬ = ⊤
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)
  = vc→ ϕ f -- def.
  × vc ϕ ξ e -- arg.
  × (∀(γ : ⟦ Γ ⟧C)(s : ⟦ S ⟧τ)(t : ⟦ T ⟧τ)
    → ϕ γ → ξ (γ , s) → ψ ((γ , s) , t) → ψ′ (γ , t))
                -- res.
vc ϕ ψ (ADDᴬ e₁ e₂) = vc ϕ (ᵏ ⊤) e₁ × vc ϕ (ᵏ ⊤) e₂
vc ϕ ψ (MINUSᴬ e₁ e₂) = vc ϕ (ᵏ ⊤) e₁ × vc ϕ (ᵏ ⊤) e₂
vc ϕ ψ (LTᴬ e₁ e₂) = vc ϕ (ᵏ ⊤) e₁ × vc ϕ (ᵏ ⊤) e₂

vc→ {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
```

Unlike the development in the book of Nipkow and Klein [20], in our language λ^A, the definition of pre deviates from wp by quite a long way. For example, the typing rule for su looks like:

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

Intuitively, when we run the wp backwards on su $\hat{e}$ with post-condition ψ, it results in $\psi[\mathrm{suc}\ \hat{e}/\nu]$. 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 rule. 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 $({}^k\top)$ $\hat{e} \neq ({}^k\top)$. For instance, if $\hat{e}$ is a function application, 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 $\mathrm{SUB^R}$, the entailment is $\phi \vDash \psi \Rightarrow \psi'$).

If we repeat the earlier example of f_0 and ex_0 in λ^A, the pre function this time indeed checks the type of the function argument, and the vc generates proof obligations about the correctness of the f_0's definition and the result of the entire function application:

```
f₀ᴬ : ∀{Γ} → Γ ⊢ᴬ `ℕ′ { (_< 2) ∘ proj₂ }⟶ `ℕ′ { (_< 4) ∘ proj₂ }
f₀ᴬ = FUNᴬ (ADDᴬ (VARᴬ top) ONEᴬ)

ex₀ᴬ : ∀{Γ} → Γ ⊢ᴬ `ℕ′
ex₀ᴬ = APPᴬ f₀ᴬ ONEᴬ

pre-ex₀ᴬ = pre {Γ = `E′} ((_< 5) ∘ proj₂) ex₀ᴬ -- 1 < 2
vc-ex₀ᴬ = vc {Γ = `E′} (ᵏ ⊤) ((_< 5) ∘ proj₂) ex₀ᴬ
{- s < 2 → s + 1 < 4 ∧      --> func definition
   s < 2 → t < 4 → t < 5   --> app result    -}
```

7.4 Meta-properties of the typechecking algorithm

We first prove (by induction on e) monotonicity of the pre and vc functions.

Lemma 7.2. *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 7.3. *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*, postcondition* ψ_1 *implies* ψ_2*, then vc* $\phi_1\ \psi_1\ e$ *implies vc* $\phi_2\ \psi_2\ e$*.*

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 7.4 (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* $\phi\ \psi \ulcorner \hat{e} \urcorner^B$ *and* $\phi\ \gamma \Rightarrow$ *pre* $\psi \ulcorner \hat{e} \urcorner^B\ \gamma$ *for any semantic environment* γ *that respects* Γ*.*

Proof. By induction on $\hat{e}$ with the help of Lemma 7.2 and Lemma 7.3. □

Corollary 7.5. *For an expression* $\Gamma \vdash_A e : T$ *in* λ^A*, if vc* $\phi\ \psi\ e$ *and* $\phi\ \gamma \Rightarrow$ *pre* $\psi\ e\ \gamma$ *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* $\ulcorner \hat{e} \urcorner^B = e$*.*

Proof. By induction on $\hat{e}$. □

Theorem 7.6 (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* $\psi\ e$*)* $\psi\ e$*, then there is a type derivation* Γ*; pre* $\psi\ e \vdash_C \hat{e} : \{\nu : T \mid \psi\}$ *such that* $\ulcorner \hat{e} \urcorner^B = e$*.*

Proof. A direct consequence of Corollary 7.5. □

8 Related Work, Future Work and Conclusion

The literature on refinement types is very rich (for example, [13, 16, 29, 30, 32], just to name a few); we find the work by Lehmann and Tanter [15] most comparable. They define the language and the logical formulae fully deeply in Coq, and assumes an oracle which 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 the oracle to construct proof terms. Knowles and Flanagan [12]'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 found in Hoare logic.

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

Language features. The language that we study in this paper is very preliminary. It does not yet support higher-order functions. How to retain the close correspondence between the refinement type system and Hoare-triple in light of higher-order functions is still unclear to us. In our formalisation, the typing judgement for function terms already breaks the triplet structure, requiring a predicate for the typing context, one for the argument type and one for the result type. As we have alluded to in the paper, if a semantic subtyping relation can be formulated, then we can possibly allow for refinement over function types, i.e. bring function types into the base type group. Otherwise, how to accommodate the predicates for the argument and result types of function objects in a Hoare-triple style formulation deserves more investigation. 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, but fighting with Agda's termination checker can be a challenge. Hoare logic style reasoning turns out to be instrumental in languages with side-effects and concurrency. How to extend the unifying paradigm to languages with such features is also an open question.

Delaying proof obligations. As we have seen in the examples, constructing a typing tree for a program requires the developer to fill in 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 of the Delay applicative functor by O'Connor [23]. It is yet to be seen how it can be applied in the construction of the typing trees in our formalisation.

Compositionality. We said in Section 7 that the λ^C language is still not fully compositional in the sense of [13]. The interpretation function $\mathscr{E}[\![\cdot]\!]_{\mathsf{Tm}}$ 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, and the implementation details of the function and the argument are hidden from the reasoning. We would like to further extend the compositionality in reasoning to other language constructs in future work.

Other program logics. Lastly, in our formalisation, we use Hoare logic as the typing rules (or program logic). There are other flavours of program logics, most notably the dual of Hoare logic – Reverse Hoare Logic [5] and Incorrectness Logic [26]. 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 present a simple yet novel Agda formalisation of refinement types on a small 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 our work is a valuable addition to the formal investigation into refinement types, and we hope this work will foster more research into machine-checked formalisations of refinement type systems, and the connection with other logical frameworks, such as Hoare logic.

References

[1] Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. 2007. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. In *Proceedings of the 5th Asian Conference on Programming Languages and Systems (APLAS'07)*. Springer-Verlag, Berlin, Heidelberg, 222–238.

[2] 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

[3] 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

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

[10] 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/

[11] 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

[12] 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

[13] 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

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

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

[16] 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

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

[18] 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

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

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

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

[22] 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

[23] 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

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

[25] 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

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

[27] 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

[28] 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

[29] 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

[30] 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

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

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

[33] 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

[34] 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