Explicit Refinement Types

JAD ELKHALEQ GHALAYINI, University of Cambridge, United Kingdom
NEEL KRISHNASWAMI, University of Cambridge, United Kingdom

We present λ_{ert}, a type theory supporting refinement types with *explicit proofs*. Instead of solving refinement constraints with an SMT solver like DML and Liquid Haskell, our system requires and permits programmers to embed proofs of properties within the program text, letting us support a rich logic of properties including quantifiers and induction. We show that the type system is sound by showing that every refined program erases to a simply-typed program, and by means of a denotational semantics, we show that every erased program has all of the properties demanded by its refined type. All of our proofs are formalised in Lean 4.

CCS Concepts: • **Theory of computation → Program verification**; **Denotational semantics**.

Additional Key Words and Phrases: Refinement Types, First Order Logic, Denotational Semantics

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

1 INTRODUCTION

Refinement typing extends an underlying type system with the ability to associate types with logical predicates on their inhabitants, constructing, for example, the type of nonempty lists or integers between three and seven. Most such systems support type dependency between the input and output types of functions, allowing us to give specifications like "the output of this function is the type of integers greater than the input." The core concept underlying type checkers for refinement types is that of logical entailment. If we have a value satisfying a predicate P and require a value satisfying Q, we require that P entails Q for our program to typecheck; this then reduces the typechecking problem to typechecking in our underlying type system plus discharging a set of entailment obligations, called our verification conditions. Given an appropriate choice of logic for our predicates, SMT solvers provide a highly effective way of automatically discharging verification conditions: this allows us to use refinement types without dealing with the bookkeeping details required by manual proofs. Combined with type and annotation inference (in which we also infer refinements), refinement types allow verifying nontrivial properties of complex programs with a minimal annotation burden, making them more appealing for use as part of a practical software development workflow — for example, the Liquid Types implementation of refinement typing for ML required a manual annotation burden of 1% to prove the DML array benchmarks safe [Rondon et al. 2008; Xi and Pfenning 1998].

However, refinement typing's reliance on automation is a double-edged sword: while it makes refinement types practical for usage in real-world contexts, it also creates a hard ceiling for expressiveness, especially if we want to use quantifiers, which would make the SMT problem undecidable.

Authors' addresses: Jad Elkhaleq Ghalayini, University of Cambridge, Department of Computer Science and Technology, Cambridge, United Kingdom, jeg74@cl.cam.ac.uk; Neel Krishnaswami, University of Cambridge, Department of Computer Science and Technology, Cambridge, United Kingdom, nk480@cl.cam.ac.uk.

2475-1421/2023/8-ART195
https://doi.org/10.1145/3607837

Rather than carefully massaging annotations into forms satisfying the particular SMT solver that is in use, it makes sense to let programmers provide manual proofs for the (hopefully, few) cases where the solver gets stuck: in utopia, humans would prove interesting things, and machines would handle the bookkeeping.

However, it is nontrivial to figure out how to add explicit proofs into a system, because the semantics of refinement type systems is subtle. So before we can add the capability to move freely between explicit proofs and automation, we need to know what manual proofs should look like! To achieve this goal, we introduce a system of explicit refinement types, λ_{ert}, in which all proofs are manual, to explore the design space. Since our proofs are entirely manual, our refinement logic can be extremely rich, and, in particular, we support full first-order logic with quantifiers.

While the presence of explicit proofs means that functions and propositions look dependently typed, λ_{ert} is not a traditional dependent type theory. In particular, we maintain the refinement discipline that "fancy types" can always be erased, leaving behind a simply-typed skeleton. Furthermore, λ_{ert} has no judgemental equality – there is no reduction in types.

Contributions

- We take a simply-typed effectful lambda calculus, λ_{stlc} (in section 5.1), and add a refinement type discipline to it, to obtain the λ_{ert} language (in Section 4).
- We support a rich logic of properties, including full first-order quantifiers, as well as ghost variables/arguments (see section 4.2). It does not rely on automated proof, and instead features *explicit* proofs of all properties.
- We show (in section 4.3) that this type system satisfies the expected properties of a type system, such as the syntactic substitution property.
- In section 5, we give a denotational semantics for both the simply-typed and refined calculi, and we prove the soundness of the intepretations. Using this, we establish semantic regularity of typing, which shows that every program respects the refinement discipline.
- We describe our mechanization of the semantics and proofs in Lean 4 in section 6. The proofs are available in the supplementary material.

2 REFINEMENT TYPES

2.1 Pragmatics of Refinement Types

Consider a simply-typed functional programming language. In many cases, the valid inputs for a function are not the entire input type but rather a subset of the input; for example, consider the classic function `head : List A → A`.

Note that we cannot define this as a total function for an arbitrary type A; we must either provide a default value, change the return type, or use some system of exceptions/partial functions.

Refinement types give us native support for types guaranteeing an invariant about their inhabitants by allowing us to associate a base type with a predicate. For example, we could represent the types of natural numbers and nonzero integers (in a Liquid-Haskell-like notation) as:

```
type Nat = {v: Int | v >= 0}
type Nonzero = {v: Int | v <> 0}
```
(1)

We could then write the type signature of division as

```
div : Int → {v: Int | v <> 0} → Int
```
(2)

By allowing dependency, where the output type of a function is allowed to depend on the value of its arguments, we can use such functionality to encode the specification of operations as well, as in

```
eq : x:Int → y:Int → {b: Bool | (x = y) = b}
```
(3)

Note that these types all use equality and inequality. This is not the *judgemental* equality of dependent type theory, but rather a *propositional* equality that can occur inside types. We could now attempt to implement a safe-division function on the natural numbers as follows

$$\begin{array}{l}\texttt{safeDiv} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat} \\ \texttt{safeDiv}\ n\ m = \mathbf{if}\ \texttt{eq}\ m\ 0\ \mathbf{then}\ 0\ \mathbf{else}\ \texttt{div}\ n\ m\end{array} \tag{4}$$

A refinement type system reduces the problem of type-checking to the question of whether a set of logical *verification conditions* holds. In this case, all of the propositions are decidable. Unfortunately, with more general specifications, checking these conditions quickly becomes undecidable. For example, if we permitted equations between arbitrary polynomials, then Hilbert's tenth problem – the solution of Diophantine equatiosn – can easily be encoded. However, by appropriately restricting the logic of assertions, we can reduce the problem of checking verification conditions to a decidable fragment of logic that, while NP-hard (or worse), can usually be solved very effectively in practice.

In particular, many languages supporting refinement types require refinements to be expressions in the quantifier-free fragment of first-order logic, with atoms restricted to carefully chosen ground theories with efficient decision procedures. In Liquid Haskell, for example, formulas in refinements are restricted to formulas in QF-UFLIA [Vazou et al. 2014]: the *quantifier-free* theory of uninterpreted functions with linear integer arithmetic. While such a system may, at first glance, seem very limited, it is possible to prove very sophisticated properties of real programs with it. In particular, with clever function definitions and termination checking (which generates a separate set of verification conditions to ensure some measure on recursive calls to a function decreases), it is even possible to perform proofs by induction in a mostly-automated fashion [Jhala and Vazou 2020].

However, it can be challenging to formulate definitions in such a limited fragment of logic, and many properties (such as monotonicity of functions or relational composition) require the use of quantifiers to be stated naturally. Unfortunately, attempts to extend solvers to support more advanced features like quantifiers usually lead to very unreliable performance.

Another issue is that systems designed around off-the-shelf solvers often do not satisfy the *de-Bruijn criterion*: their trusted code-base, rather than being restricted to a small, easily-trusted kernel, instead will often consist of an entire solver and associated tooling! Unfortunately, SMT solvers are incredibly complex pieces of software and hence are very likely to have soundness bugs, as recent research on fuzzing state-of-the-art solvers like CVC4 and Z3 for bugs tends to show [Winterer et al. 2020].

In our work, we build a refinement type system, which replaces the use of automated solvers with explicit proofs. This permits (at the price of writing proofs) the free use of quantifiers in propositions, while also having a trusted codebase small enough to be formally verified.

2.2 Semantics of Refinement Types

To understand what refinement types are semantically, it is worth recalling that there are two main styles of doing denotational semantics, which Reynolds dubbed "intrinsic" and "extrinsic" [Reynolds 2003].

The intrinsic style is the usual style of categorical semantics – we find some category where types and contexts are objects, and a term $\Gamma \vdash e : A$ is interpreted as an element of the hom-set $\mathrm{Hom}(\Gamma, A)$. (Dependency does make things more complicated technically, but not in any conceptually essential fashion.) In the intrinsic style, only well-typed terms have denotations, and ill-typed terms are grammatically ill-formed and do not have a semantics at all. Another way of putting this is that intrinsic semantics interprets *typing derivations* rather than *terms*.

In the extrinsic style, the interpretation function is defined upon raw terms, and ill-typed programs *do* get a semantics. Types are then defined as retracts of the underlying semantic object interpreting the raw terms. A good example of this style of semantics is Milner's proof of type safety for ML in [Milner 1978]: he gave a denotational semantics for an untyped lambda calculus, and then gave an interpretation of types by logical relations over this untyped calculus, which let him use the fundamental lemma to extract type safety. Logical relations (and realizabilty methods in general) are typical examples of extrinsic semantics. Normally type theorists elide the distinction between these two styles of semantics, because we prove coherence: that the same term cannot have two derivations with different semantics. This is called the "bracketing theorem" in Reynolds [2003] and Melliès and Zeilberger [2015].

Refinement types puts these two styles of semantics together in a different way. We start with a unrefined, base language with an intrinsic semantics, and then define a *second* system of types as a family of retracts over the base types. In λ_{ert}, we use a monadic lambda-calculus as the base language, and define our system of refinements over this calculus, with the refinements structured as a fibration over the base calculus. We deliberately avoided foregrounding the categorical machinery in our paper to make it more accessible, though we do direct readers to Melliès and Zeilberger [2015]'s POPL paper (which deeply influenced our design).

Another, perhaps more familiar, instance of this framework are Hoare logic and separation logic, which can be understood as refinement type systems over a base "uni-typed" imperative language. In fact, the distinction in Hoare logic between "logical variables" (which appear in specifications) and program variables is (in semantic terms) precisely the same as the distinction between logical and computational terms in λ_{ert} – which we will discuss further in the next section.

3 EXPLICIT REFINEMENT TYPES

In this section, we introduce our system of *explicit refinement types*, λ_{ert}, which we construct by enriching the simply-typed lambda calculus with proofs, intersection types, and union types. We recover a computational interpretation of terms in our calculus by recursively erasing the logical information added, yielding back simply-typed λ_{stlc} terms. The presence of proofs allows us to track logical facts, and by pairing a term with a proof of its property, form subset types. We also support a general form of intersection and union type, to allow us to pass around terms used only in proofs in a way that is guaranteed to be *computationally irrelevant*, which is both a significant performance concern and allows type signatures to more clearly express the programmer's intent. For example, consider the following definition of a vector type:

$$\mathsf{Vec}\ A\ n \equiv \{\ell : \mathsf{List}\ A \mid \mathsf{List.len}\ \ell = n\} \tag{5}$$

In particular, we define a vector of length n as a list paired with the information that the list has length n. That is, a vector is a subset type: a base type $\mathsf{List}\ A$ paired with a proposition $P(\ell)$ (here $\mathsf{len}\ \ell = n$), which we will interpret as containing all elements ℓ of the base type satisfying $P(\ell)$. In general, we write such types as $\{x : X \mid P(x)\}$, and introduce them with the form $\{e, p\}$, where e is of type X and p is a proof of $P(e)$. We define a length function on vectors as follows:

$$\begin{array}{llll} \mathsf{Vec.len} & : & \forall n : \mathbb{N}, & \mathsf{Vec}\ A\ n \to & \mathbb{N} \\ \mathsf{Vec.len} & \equiv & \hat{\lambda} \| n : \mathbb{N} \|, & \lambda v : \mathsf{Vec}\ A\ n, & \mathsf{let}\ \{\ell, p\} = v\ \mathsf{in}\ \mathsf{List.len}\ \ell \end{array} \tag{6}$$

Let us break this definition down, starting from the signature. Vec.len begins by universally quantifying over a natural number n with the quantifier $\forall n : \mathbb{N}$, and then has a function type $\mathsf{Vec}\ A\ n \to \mathbb{N}$. We can interpret this as saying that for every n, Vec.len takes vectors of length n to a natural number.

The $\forall$ quantifier in the type of Vec.len behaves like the quantifiers in ML-style polymorphism, rather than the pi-type of dependent type theory. It indicates that the *same* function can handle vectors of any length, with no explicit branching on the length. In contrast, a pi-type can lets the function body compute with the natural number argument.

An explicit binding, written $\lambda \| n : \mathbb{N} \|$, represents this quantifier in the actual definition; we call a variable binding surrounded by double bars (e.g., $\| x : A \|$) a *ghost binding*. Moving inwards, we have a function type with input Vec A n and output $\mathbb{N}$; this corresponds to the lambda-expression "$\lambda v : \mathsf{Vec}\ A\ n, E$" in the definition, where $E \equiv \mathsf{let}\ \{\ell, \_\} = v\ \mathsf{in}\ \mathsf{len}\ \ell$ is an expression of type $\mathbb{N}$. Breaking down E, we must first explicitly destructure our vector v into its components, a list ℓ and a proof, p that ℓ is of length n. Unlike in refinement type systems like DML and Liquid Haskell, this is explicit, with no entailment-based subtyping.

The definition in equation 6 ignores the proof component of the let-binding and hence the refinement information carried by our type system. One way to use the proof information would be to have a definition for Vec.len, which promises to return an integer equal to the length.

$$\begin{array}{llll} \mathsf{Vec.len}' & : & \forall n : \mathbb{N}, & (v : \mathsf{Vec}\ A\ n) \rightarrow \quad \{x : \mathbb{N} \mid x = n\} \\ \mathsf{Vec.len}' & \equiv & \hat{\lambda} \| n : \mathbb{N} \|, & \lambda v : \mathsf{Vec}\ A\ n, \quad \mathsf{let}\ \{\ell, p\} = v\ \mathsf{in}\ \{\mathsf{List.len}\ \ell, p\} \end{array} \tag{7}$$

However, one helpful feature of λ_{ert} is that even in this case, we can prove facts about our definitions by writing freestanding proofs about them rather than cramming every possible fact we could want into our type signature. For example, we could give a proof a proposition that the definition in equation 6 is correct as follows:

$$\begin{array}{ll} \mathsf{Vec.len\_def} : & \forall n : \mathbb{N}, \forall v : \mathsf{Vec}\ A\ n, \mathsf{Vec.len}\ \|n\|\ v = n \\ \mathsf{Vec.len\_def} \equiv & \hat{\lambda} \| n : \mathbb{N} \|, \hat{\lambda} \| v : \mathsf{Vec}\ A\ n \|, \mathsf{let}\ \{\ell, p\} = v\ \mathsf{in} \\ & \mathsf{trans}[\mathsf{Vec.len}\ \|n\|\ \{\ell, p\} \\ & \qquad =(\beta_{\mathsf{ir}})\ (\lambda v, \mathsf{let}\ \{\ell, \_\} = v\ \mathsf{in}\ \mathsf{List.len}\ \ell)\{\ell, p\} \\ & \qquad =(\beta_{\mathsf{ty}})\ (\mathsf{let}\ \{\ell, \_\} = \{\ell, p\}\ \mathsf{in}\ \mathsf{List.len}\ \ell) \\ & \qquad =(\beta_{\mathsf{set}})\ \ \mathsf{List.len}\ \ell \\ & \qquad =(p)\ \ n] \end{array} \tag{8}$$

Let us break this definition down, again starting with the signature. We quantify over both the length $n : \mathbb{N}$ and the vector $v : \mathsf{Vec}\ A\ n$, and then assert the equality proposition $\mathsf{Vec.len}\ \|n\|\ v = n$. In the proofs, both universal quantifiers are introduced by ghost lambdas "$\hat{\lambda} \| n : \mathbb{N} \|$" and "$\hat{\lambda} \| v : \mathsf{Vec}\ A\ n \|$". However, we use $\hat{\lambda}$ instead of λ because we are proving a universally quantified *proposition* rather than forming a *term* of intersection type. The proof of equality is a term of the form trans[...], which represents an Agda-style syntax sugar for equational reasoning. In particular, if $\mathsf{trans}_{p,q,r} : p = q \rightarrow q = r \rightarrow p = r$ is the transitivity rule, then we have the desugaring

$$\mathsf{trans}[x_0 =(p_0)\ x_1 =(p_1)\ x_2\ \ldots\ x_n =(p_n)\ x_{n+1}] \equiv \mathsf{trans}_{x_0,x_1,x_n}\ p_0(\mathsf{trans}_{x_1,x_2,x_n}\ p_1\ (\ldots)) \tag{9}$$

where each p_i is evidence of the proposition $x_i = x_{i+1}$.

Examining the proof of equation 5, we see that every piece of evidence used except one is of the form "$\beta_{\mathsf{something}}$". These are explicit β-reduction proofs, which are necessary since our calculus does not include a notion of judgemental equality: type equality is simply α-equivalence. [1] While this dramatically simplifies the meta-theory, this can make even simple proofs very long. In our examples, we implement the pattern of repeatedly applying a β-reduction as syntactic sugar. Writing it as β (pronounced "by beta"), we get the much simpler proof

$$\begin{array}{ll} \mathsf{Vec.len\_def} & : \forall n : \mathbb{N}, \forall v : \mathsf{Vec}\ A\ n, \mathsf{len}\ v = n \\ \mathsf{Vec.len\_def} & \equiv \hat{\lambda} \| n : \mathbb{N} \|, \hat{\lambda} \| v : \mathsf{Vec}\ A\ n \|, \\ & \qquad \mathsf{let}\ \{\ell, p\} = v\ \mathsf{in}\ \mathsf{trans}[\mathsf{len}\ n\ \{\ell, p\} =(\beta)\ \mathsf{len}\ \ell =(p)\ n] \end{array} \tag{10}$$

[1]The actual β-reduction rules in our calculus, e.g. β_{ty}, require explicit annotations to be unambiguous. We omit these here for space and clarity.

However, these definitions raise a question: why not simply write

$$\mathsf{Vec.len} \equiv \lambda \|n : \mathbb{N}\|, \lambda v : \mathsf{Vec}\ A\ n, n \tag{11}$$

The problem with the above definition is that n is a ghost variable, indicating we may use it in propositions and proofs but not generally in terms. This is a critical distinction to be able to make: the specification of a program may involve values which we do not want to manipulate at runtime. For example, the correctness proof of a sorting routine might use an inductive datatype of permutations, elements of which could potentially be much larger than the list itself. By making a distinction between ghost and computational variables, we can define an efficient erasure of refined terms (which contain proofs) to simply-typed terms (which do not). For example, we can erase the signature in equation 7 into a simple type:

$$\begin{aligned}
&|\forall n : \mathbb{N}, (v : \mathsf{Vec}\ A\ n) \to \{x : \mathbb{N} \mid x = n\}| && (12)\\
&= \mathbf{1} \to |\{\ell : \mathsf{List}\ A \mid \mathsf{len}\ \ell = n\} \to \{x : \mathbb{N} \mid x = n\}| && \text{(unfold, erase quantified variable } n)\\
&= \mathbf{1} \to |\{\ell : \mathsf{List}\ A \mid \mathsf{len}\ \ell = n\}| \to |\{x : \mathbb{N} \mid x = n\}| && \text{(erasure distributes over function types)}\\
&= \mathbf{1} \to |\mathsf{List}\ A| \to |\mathbb{N}| && \text{(erase subset types to erased base types)}\\
&= \mathbf{1} \to \mathsf{List}\ |A| \to \mathbb{N} && \text{(list erases to list of erased type, } \mathbb{N} \text{ erases to } \mathbb{N})
\end{aligned}$$

We can then erase the definition as follows:

$$\begin{aligned}
&|\lambda \|n : \mathbb{N}\|, \lambda v : \mathsf{Vec}\ A\ n, \mathsf{let}\ \{\ell, \_\} = v\ \mathsf{in}\ \mathsf{len}\ \ell| && (13)\\
&= \lambda n : \mathbf{1}, |\lambda v : \{\ell : \mathsf{List}\ A \mid \mathsf{len}\ \ell = n\}, \mathsf{let}\ \{\ell, \_\} = v\ \mathsf{in}\ \mathsf{len}\ \ell| && \text{(unfold, erase quantified variable } n)\\
&= \lambda n : \mathbf{1}, \lambda v : |\{\ell : \mathsf{List}\ A \mid \mathsf{len}\ \ell = n\}|, |\mathsf{let}\ \{\ell, \_\} = v\ \mathsf{in}\ \mathsf{len}\ \ell| && \text{(erasure distributes over binders)}\\
&= \lambda n : \mathbf{1}, \lambda v : |\mathsf{List}\ A|, |\mathsf{let}\ \{\ell, \_\} = v\ \mathsf{in}\ \mathsf{len}\ \ell| && \text{(erase subset types to erased base types)}\\
&= \lambda n : \mathbf{1}, \lambda v : \mathsf{List}\ |A|, |\mathsf{let}\ \{\ell, \_\} = v\ \mathsf{in}\ \mathsf{len}\ \ell| && \text{(list erases to list of erased type)}\\
&= \lambda n : \mathbf{1}, \lambda v : \mathsf{List}\ |A|, \mathsf{let}\ \ell = |v|\ \mathsf{in}\ |\mathsf{len}\ \ell| && \text{(erase distributes over let)}\\
&= \lambda n : \mathbf{1}, \lambda v : \mathsf{List}\ |A|, \mathsf{let}\ \ell = v\ \mathsf{in}\ |\mathsf{len}\ \ell| && \text{(variables erase to themselves)}\\
&= \lambda n : \mathbf{1}, \lambda v : \mathsf{List}\ |A|, \mathsf{let}\ \ell = v\ \mathsf{in}\ |\mathsf{len}|\ |\ell| && \text{(erasure distributes over application)}\\
&= \lambda n : \mathbf{1}, \lambda v : \mathsf{List}\ |A|, \mathsf{let}\ \ell = v\ \mathsf{in}\ \mathsf{len}\ \ell : \mathbf{1} \to \mathsf{List}\ |A| \to \mathbb{N} && \text{(variables, len erase to themselves)}
\end{aligned}$$

At the type level, we erase any dependency information, leaving us with simple types. Propositions are all either wholly erased or erased to the unit type. At the term level, erasure essentially consists of recursively erasing ghost variables and proofs into units and (in the case of proofs of falsehood) error stops, yielding a well-typed term in the simply-typed lambda calculus extended with an error stop effect (i.e., the exception monad). We will prove in section 5 that the produced terms are always well-typed.

As expected, we see that erasure sends base types like $\mathbf{1}$ and $\mathbb{N}$, as well as their literals like $()$ or 0, to themselves. Erasure distributes over (dependent) function types, i.e., $|(a : A) \to B(a)| = |A| \to |B(a)|$, and correspondingly is recursively applied to the argument type and result of a lambda function as follows: $|\lambda a : A, b| = \lambda a : |A|, |b|$. Subset types are erased to the erasure of their base type, i.e., we have $|\{x : X \mid P(x)\}| = |X|$. Erasure of the formation and elimination rules for subset types is likewise as expected, with $|\{x, p\}| = |x|$ and $|\mathsf{let}\ \{x, p\} = e\ \mathsf{in}\ e'| = \mathsf{let}\ x = |e|\ \mathsf{in}\ |e'|$. Similarly, universal quantifiers are erased to the unit type $\mathbf{1}$ as follows: $|\forall a : A, B(a)| = \mathbf{1} \to |B(a)|$. Correspondingly, we erase the introduction and elimination rules for intersection types like so: $|\lambda \|a : A\|, b| = \lambda\_ : \mathbf{1}, |b|$, $|f\ \|a\|| = |f|\ ()$. In general, propositions and ghosts in value types (such as subsets) are erased completely, whereas propositions and ghosts in function-style types (such as intersection types) are erased to units to avoid problems with the eager evaluation order of λ_{stlc}. (See section 7 for a more detailed discussion.)

Returning to the original question, erasing the definition in equation 11 yields the term $\lambda n : \mathbf{1}, \lambda v : \mathsf{List}\ |A|, n$, which is not well-typed at $\mathbf{1} \longrightarrow \mathsf{List}\ |A| \longrightarrow \mathbb{N}$. Another way of thinking about this is that if Vec.len took the length n as a computational argument, we would never need to call the function – we would have to have the length in hand to call Vec.len in the first place. However, in our setting, vectors are merely refined lists, which erase into raw lists. A raw list does not carry its length n; but n is a well-defined property of its specification. Since the specification value n gets erased to a unit, then a program that wants to compute the length must traverse the list – i.e., in equation 6 we call List.len.

Another advantage of having explicit proofs as part of a refinement type system is the ability to reuse previous theorems and perform proofs by induction. For a simple example of this, consider the following definition of addition for natural numbers:

$$n + m \equiv (\mathsf{natrec}\ n\ (\lambda x, x)\ (\|\mathsf{succ}\ \_\|, f \mapsto \lambda x, \mathsf{succ}\ (f\ x)))\ m : \mathbb{N} \tag{14}$$

natrec is the eliminator for the natural numbers, which is defined essentially by iteration, with typing rule Natrec. The eliminator has the standard behavior, given by reduction rules β_{zero} and β_{succ}, which essentially amount to substituting z into s recursively n times. We can then use these axioms to prove that zero is a left-identity by β-reduction, simply writing $\mathsf{zero}_{\mathsf{left}} : (\forall n : \mathbb{N}, 0 + n = n) \equiv \hat{\lambda}\|n : \mathbb{N}\|, \beta$. In contrast, we need induction to prove that zero is a right-identity. To perform induction, we introduce the ind eliminator for natural numbers, essentially the propositional version of natrec, with typing rule Ind. We may then write

$$\begin{aligned}
&\mathsf{zero}_{\mathsf{right}} : (\forall n : \mathbb{N}, n + 0 = n) \equiv \\
&\quad \hat{\lambda}\|n : \mathbb{N}\|, \mathsf{ind}[x \mapsto x + 0 = x]\ n\ \beta \\
&\qquad\qquad (\mathsf{succ}\ n, u \mapsto \mathsf{trans}[(\mathsf{succ}\ n) + 0 =(\beta)\ \mathsf{succ}\ (n + 0) =(u)\ \mathsf{succ}\ n]))
\end{aligned} \tag{15}$$

Induction lets us prove many arithmetic facts without baking them into the type system. For example, we can prove that addition is commutative as follows (see figure 1 for helpers), where symm and congr are proofs that equality is symmetric and transitive respectively:

$$\begin{aligned}
&\mathsf{add}_{\mathsf{comm}} : (\forall n, m : \mathbb{N}, n + m = m + n) \equiv \hat{\lambda}\|n : \mathbb{N}\|, \\
&\quad \mathsf{ind}[x \mapsto \forall m : \mathbb{N}, x + m = m + x]\ n\ \mathsf{zero}_{\mathsf{comm}} \\
&\qquad (\mathsf{succ}\ n, u \mapsto \mathsf{trans}[\mathsf{succ}\ n + m \\
&\qquad\quad =(\beta)\ \mathsf{succ}\ (n + m) \\
&\qquad\quad =(\mathsf{congr}\ u)\ \mathsf{succ}\ (m + n) \\
&\qquad\quad =(\beta)\ \mathsf{succ}\ m + n \\
&\qquad\quad =(\mathsf{symm}\ (\mathsf{succ}_{\mathsf{comm}}\ \|\mathsf{m}\|\ \|\mathsf{n}\|))\ m + \mathsf{succ}\ n])
\end{aligned} \tag{16}$$

Note the ability to reuse theorems we have proved previously. Another advantage of explicit proofs is that we do not need to encode even fundamental facts into our core calculus since we can prove them from a small core of base axioms. Minimizing the number of axioms simplifies the implementation of the type-checker and reduces the size of the trusted codebase while allowing the programmer to effectively write refinements and proofs using facts that the language designer may not have considered.

$$
\begin{aligned}
&\mathsf{zero}_{\mathsf{comm}} : (\forall n : \mathbb{N}, 0 + n = n + 0) \equiv \hat{\lambda}\|n : \mathbb{N}\|, \mathsf{trans}[0 + n =(\beta)\ n =(\mathsf{zero}_{\mathsf{right}}\ \|n\|)\ n + 0] \\
&\mathsf{succ}_{\mathsf{right}} : (\forall n, m : \mathbb{N}, n + \mathsf{succ}\ m = \mathsf{succ}\ (n + m)) \equiv \\
&\qquad \hat{\lambda}\|n, m : \mathbb{N}\|, \mathsf{ind}[x \mapsto x + (\mathsf{succ}\ m) = \mathsf{succ}\ (x + m)]\ n\ \beta \\
&\qquad\qquad\qquad (\mathsf{succ}\ n, u \mapsto \mathsf{trans}[\mathsf{succ}\ n + \mathsf{succ}\ m \\
&\qquad\qquad\qquad\quad =(\beta)\ \mathsf{succ}\ (n + \mathsf{succ}\ m) \\
&\qquad\qquad\qquad\quad =(u)\ \mathsf{succ}\ (\mathsf{succ}\ (n + m)) \\
&\qquad\qquad\qquad\quad =(\beta)\ \mathsf{succ}\ (\mathsf{succ}\ n + m)])) \\
&\mathsf{succ}_{\mathsf{comm}} : (\forall n : \mathbb{N}, n + \mathsf{succ}\ m = \mathsf{succ}\ n + m) \equiv \\
&\qquad \hat{\lambda}\|n : \mathbb{N}\|, \mathsf{trans}[n + \mathsf{succ}\ m =(\mathsf{succ}_{\mathsf{right}}\ \|n\|\ \|m\|)\ \mathsf{succ}(n + m) =(\beta)\ \mathsf{succ}\ n + m]
\end{aligned}
$$

Fig. 1. Helper definitions for equation 16

Judgment	Meaning
Γ ok	Γ is a well-formed context
$\Gamma \vdash A$ ty	A is a well-formed type in Γ
$\Gamma \vdash \varphi$ pr	φ is a well-formed proposition in Γ
$\Gamma \vdash a : A$	a may consistently be assigned type A in Γ
$\Gamma \vdash p : \varphi$	p is a proof of φ in Γ

Fig. 2. λ_{ert} typing judgements

4 FORMALIZATION

We give λ_{ert}'s grammar and typing rules in sections 4.1 and 4.2. We then give proofs of some expected metatheoretic properties, such as substitution and regularity, in section 4.3.

4.1 Grammar

The grammar for λ_{ert} (given in the appendix) consists of four separate syntactic categories: types A, propositions φ, terms a, and proofs p. We denote the set of (syntactically) well-formed terms living in each by Type, Prop, Term, and Proof, respectively. We may then define a typing context Γ as a list of computational variables $x : A$, ghost variables $\|x : A\|$, and propositional variables $u : \varphi$, as in figure 3.

Ghost variables only appear within proofs, types, and propositions, whereas computational variables can occur anywhere, including both computational and logical terms. Typing contexts are telescopic, so types and propositions appearing later in the context may depend on previously defined variables. We use these syntactic categories to state the shape of λ_{ert}'s typing judgments in figure 2. A context is *well-formed* if the types of each of its variables are well-formed in the context made up of all previously defined variables, with the context well-formedness rules in figure 3. The presence of both computational and ghost variables means that our contexts have additional structural properties beyond the usual ones such as weakening and exchange. Since a computational variable can be used in more places than a ghost variable, we introduce the concept of an *upgrade*.

$$\frac{}{\cdot\ \mathsf{ok}} \qquad \frac{\Gamma\ \mathsf{ok} \quad \Gamma \vdash A\ \mathsf{ty}}{\Gamma, x : A\ \mathsf{ok}} \qquad \frac{\Gamma\ \mathsf{ok} \quad \Gamma \vdash A\ \mathsf{ty}}{\Gamma, \|x : A\|\ \mathsf{ok}} \qquad \frac{\Gamma\ \mathsf{ok} \quad \Gamma \vdash \varphi\ \mathsf{pr}}{\Gamma, u : \varphi\ \mathsf{ok}}$$

Fig. 3. λ_{ert} context well-formedness rules

When we upgrade a context, some of the ghost variables can be replaced by computational variables with the same name and type. So, for example, the context $\|x : A\|, y : B$ can be upgraded to $x : A, y : B$. In figure 4, we formalise this with the rules of the judgment $\Delta \leq \Gamma$, which reads "Δ upgrades Γ".

$$\frac{\Delta \leq \Gamma}{\Delta, x : A \leq \Gamma, \|x : A\|} \qquad \frac{\Delta \leq \Gamma}{\Delta, H \leq \Gamma, H} \qquad \frac{}{\cdot \leq \cdot}$$

Fig. 4. λ_{ert} context upgrade rules, where H ranges over hypotheses $x : A$, $\|x : A\|$, $p : \varphi$

We now define the *upgrade* of Γ, written $\Gamma^{\uparrow}$, to be the context with all ghost variables in Γ replaced by term variables, that is,

$$\cdot^{\uparrow} = \cdot \qquad (\Gamma, \|x : A\|)^{\uparrow} = \Gamma^{\uparrow}, x : A, \qquad (\Gamma, x : A)^{\uparrow} = \Gamma^{\uparrow}, x : A, \qquad (\Gamma, u : \varphi)^{\uparrow} = \Gamma^{\uparrow}, u : \varphi \tag{17}$$

Note that $\Gamma^{\uparrow} \leq \Gamma$ (but $\Gamma \leq \Gamma^{\uparrow}$ iff Γ contains no ghost variables). A context Δ which upgrades Γ types more terms than Γ, since we may use a computational variable anywhere a ghost variable is expected, but not vice versa. In particular, we may prove the following lemma:

Lemma 4.1 (Upgrade). *Given contexts* $\Delta \leq \Gamma$,

- *If* $\Gamma \vdash \varphi$ pr, *then* $\Delta \vdash \varphi$ pr. *In particular, if* $\Gamma \vdash \varphi$ pr, *then* $\Gamma^{\uparrow} \vdash \varphi$ pr.
- *If* $\Gamma \vdash A$ ty, *then* $\Delta \vdash A$ ty. *In particular, if* $\Gamma \vdash \varphi$ ty, *then* $\Gamma^{\uparrow} \vdash \varphi$ ty.
- *If* $\Gamma \vdash p : \varphi$, *then* $\Delta \vdash p : \varphi$. *In particular, if* $\Gamma \vdash p : \varphi$, *then* $\Gamma^{\uparrow} \vdash p : \varphi$.
- *If* $\Gamma \vdash a : A$, *then* $\Delta \vdash a : A$. *In particular, if* $\Gamma \vdash a : A$, *then* $\Gamma^{\uparrow} \vdash a : A$.
- *If* Γ ok, *then* Δ ok. *In particular, if* Γ ok, *then* $\Gamma^{\uparrow}$ ok.

Since the only difference between computational and ghost variables is that ghosts can't be used in computational terms, this distinction does not matter for proofs, or for proposition- and type-well-formedness. Formally:

Lemma 4.2 (Downgrade). *Given contexts* $\Delta \leq \Gamma$,

- *If* $\Delta \vdash \varphi$ pr, *then* $\Gamma \vdash \varphi$ pr. *In particular, if* $\Gamma^{\uparrow} \vdash \varphi$ pr, *then* $\Gamma \vdash \varphi$ pr.
- *If* $\Delta \vdash A$ ty, *then* $\Gamma \vdash A$ ty. *In particular, if* $\Gamma^{\uparrow} \vdash \varphi$ ty, *then* $\Gamma \vdash \varphi$ ty.
- *If* $\Delta \vdash p : \varphi$, *then* $\Gamma \vdash p : \varphi$. *In particular, if* $\Gamma^{\uparrow} \vdash p : \varphi$, *then* $\Gamma \vdash p : \varphi$.
- *If* Δ ok, *then* Γ ok. *In particular, if* $\Gamma^{\uparrow}$ ok, *then* Γ ok.

4.2 Typing Rules

The type formation rules for λ_{ert} are collected in figure 5, the term formation rules in figure 7, the proposition formation rules in figure 6, and the proof rules and axioms in figures 8 and 9 respectively.

$$\frac{\Gamma \vdash A\ \mathsf{ty} \quad \Gamma, x : A \vdash B\ \mathsf{ty}}{\Gamma \vdash (x : A) \to B\ \mathsf{ty}}\ \textsf{Fn-WF} \qquad \frac{\Gamma \vdash A\ \mathsf{ty} \quad \Gamma, x : A \vdash B\ \mathsf{ty}}{\Gamma \vdash (x : A) \times B\ \mathsf{ty}}\ \textsf{Pair-WF}$$

$$\frac{\Gamma \vdash A\ \mathsf{ty} \quad \Gamma, x : A \vdash B\ \mathsf{ty}}{\Gamma \vdash \forall x : A, B\ \mathsf{ty}}\ \textsf{Intr-WF} \qquad \frac{\Gamma \vdash A\ \mathsf{ty} \quad \Gamma, x : A \vdash B\ \mathsf{ty}}{\Gamma \vdash \exists x : A, B\ \mathsf{ty}}\ \textsf{Union-WF}$$

$$\frac{\Gamma \vdash \varphi\ \mathsf{pr} \quad \Gamma, u : \varphi \vdash A\ \mathsf{ty}}{\Gamma \vdash (u : \varphi) \Rightarrow A\ \mathsf{ty}}\ \textsf{Pre-WF} \qquad \frac{\Gamma \vdash A\ \mathsf{ty} \quad \Gamma, x : A \vdash \varphi\ \mathsf{pr}}{\Gamma \vdash \{x : A \mid \varphi\}\ \mathsf{ty}}\ \textsf{Set-WF}$$

$$\frac{}{\Gamma \vdash \mathbf{1}\ \mathsf{ty}}\ \textsf{Unit-WF} \qquad \frac{\Gamma \vdash A\ \mathsf{ty} \quad \Gamma \vdash B\ \mathsf{ty}}{\Gamma \vdash A + B\ \mathsf{ty}}\ \textsf{Coprod-WF} \qquad \frac{}{\Gamma \vdash \mathbb{N}\ \mathsf{ty}}\ \textsf{Nats-WF}$$

Fig. 5. λ_{ert} Type Well-formedness

$$\frac{\Gamma \vdash \varphi\ \mathsf{pr} \quad \Gamma, u : \varphi \vdash \psi\ \mathsf{pr}}{\Gamma \vdash (u : \varphi) \Rightarrow \psi\ \mathsf{pr}}\ \textsf{Imp-WF} \qquad \frac{\Gamma \vdash \varphi\ \mathsf{pr} \quad \Gamma \vdash \psi\ \mathsf{pr}}{\Gamma \vdash \varphi \vee \psi\ \mathsf{pr}}\ \textsf{Or-WF} \qquad \frac{\Gamma \vdash \varphi\ \mathsf{pr} \quad \Gamma, u : \varphi \vdash \psi\ \mathsf{pr}}{\Gamma \vdash (u : \varphi) \wedge \psi\ \mathsf{pr}}\ \textsf{And-WF}$$

$$\frac{\Gamma \vdash A\ \mathsf{ty} \quad \Gamma, x : A \vdash \varphi\ \mathsf{pr}}{\Gamma \vdash \forall x : A, \varphi\ \mathsf{pr}}\ \textsf{Univ-WF} \qquad \frac{\Gamma \vdash A\ \mathsf{ty} \quad \Gamma, x : A \vdash \varphi\ \mathsf{pr}}{\Gamma \vdash \exists x : A, \varphi\ \mathsf{pr}}\ \textsf{Exists-WF}$$

$$\frac{\Gamma \vdash A\ \mathsf{ty} \quad \Gamma^{\uparrow} \vdash a : A \quad \Gamma^{\uparrow} \vdash b : A}{\Gamma \vdash a =_A b\ \mathsf{pr}}\ \textsf{Eq-WF} \qquad \frac{}{\Gamma \vdash \top\ \mathsf{pr}}\ \textsf{True-WF} \qquad \frac{}{\Gamma \vdash \bot\ \mathsf{pr}}\ \textsf{False-WF}$$

Fig. 6. λ_{ert} Proposition Well-formedness

4.2.1 Equality. The heart of the λ_{ert} type refinement system is the equality proposition $a =_A b$, which is verified by proofs that $a = b$ where a and b are interpreted as elements of the type A. This has formation rule Eq-WF, which checks that a and b are well-typed in an upgraded context. Because equality is a mathematical proposition, we may use ghost variables freely. Furthermore, note that we can consider equalities between terms of any type, including terms of higher type such as function types. This has often been challenging to support with refinement types (see Vazou and Greenberg [2022] for a detailed discussion), but is unproblematic with λ_{ert}.

The introduction rule is the standard reflexivity axiom, Rfl, and equalities may be eliminated via substitution using the rule Subst. The substitution eliminator is powerful enough to prove the other equality axioms, such as, for example, transitivity:

$$\frac{\Gamma \vdash p : a =_A b \quad \Gamma \vdash q : b =_A c}{\mathsf{trans}\ p\ q \equiv \mathsf{subst}[x \mapsto a =_A x][b][c]\ q\ p : a =_A c} \tag{18}$$

Type equality λ_{ert} is just α-equivalence, and so any equations which would have come via judgemental equality in a dependent type theory must be expressed as equality axioms. For example, beta-reduction for functions is expressed via the axiom β_{ty}, and there are similar rules for each of the type constructors in the language. Furthermore, because proofs are computationally irrelevant, they support an extensionality principle: the axiom Ir-Pr lets us replace any proof p with any other proof q.

4.2.2 Type Structure. λ_{ert} has (refinements of) the usual type constructors of the simply-typed lambda calculus, such as functions, pairs, sum types, and datatypes like natural numbers, as well as type constructors specific to refinement types such as subset types, generalised intersections and unions, and preconditions.

$$\frac{}{\Gamma, x : A \vdash x : A}\,\textsf{Var} \qquad \frac{\Gamma \vdash p : \bot}{\Gamma \vdash \mathsf{absurd}\ p : A}\,\textsf{Absurd} \qquad \frac{}{\Gamma \vdash 0 : \mathbb{N}}\,\textsf{Zero} \qquad \frac{}{\Gamma \vdash \mathsf{succ} : \mathbb{N} \to \mathbb{N}}\,\textsf{Succ}$$

$$\frac{}{\Gamma \vdash () : \mathbf{1}}\,\textsf{Unit} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A, e : (x : A) \to B}\,\textsf{Lam} \qquad \frac{\Gamma \vdash l : (x : A) \to B \quad \Gamma \vdash r : A}{\Gamma \vdash l\ r : [r/x]B}\,\textsf{App}$$

$$\frac{\Gamma \vdash l : A \quad \Gamma \vdash r : [l/x]B}{\Gamma \vdash (l, r) : (x : A) \times B}\,\textsf{Pair} \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash \mathsf{inl}\ e : A + B}\,\textsf{Inl} \qquad \frac{\Gamma \vdash e : B}{\Gamma \vdash \mathsf{inr}\ e : A + B}\,\textsf{Inr}$$

$$\frac{\Gamma \vdash e : (x : A) \times B \quad \Gamma, z : (x : A) \times B \vdash C\ \mathsf{ty} \quad \Gamma, x : A, y : B \vdash e' : [(x, y)/z]C}{\Gamma \vdash \mathsf{let}\ (x, y) : (x : A) \times B = e\ \mathsf{in}\ e' : [e/z]C}\,\textsf{Let-Pair}$$

$$\frac{\Gamma, x : A + B \vdash C\ \mathsf{ty} \quad \Gamma \vdash e : A + B \quad \Gamma, y : A \vdash l : [\mathsf{inl}\ y/x]C \quad \Gamma, z : B \vdash r : [\mathsf{inr}\ z/x]C}{\Gamma \vdash \mathsf{cases}\ [x \mapsto C]\ e\ (\mathsf{inl}\ y \mapsto l)\ (\mathsf{inr}\ z \mapsto r) : [e/x]C}\,\textsf{Cases}$$

$$\frac{\Gamma, u : \varphi \vdash e : A}{\Gamma \vdash \lambda u : \varphi.e : (u : \varphi) \Rightarrow A}\,\textsf{Lam-Pr} \qquad \frac{\Gamma \vdash f : (u : \varphi) \Rightarrow A \quad \Gamma \vdash p : \varphi}{\Gamma \vdash f\ p : [p/x]A}\,\textsf{App-Pr}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash p : [a/x]\varphi}{\Gamma \vdash \{a, p\} : \{x : A \mid \varphi\}}\,\textsf{Set} \qquad \frac{\Gamma^{\uparrow} \vdash a : A \quad \Gamma \vdash b : [a/x]B}{\Gamma \vdash (\|a\|, b) : \exists a : A, B}\,\textsf{Pair-Ir}$$

$$\frac{\Gamma \vdash e : \{x : A \mid \varphi\} \quad \Gamma, z : \{x : A \mid \varphi\} \vdash C\ \mathsf{ty} \quad \Gamma, x : A, u : \varphi \vdash e' : [\{x, u\}/z]C}{\Gamma \vdash \mathsf{let}\ \{x, u\} : \{x : A \mid \varphi\} = e\ \mathsf{in}\ e' : [e/z]C}\,\textsf{Let-Set}$$

$$\frac{\Gamma, \|x : A\| \vdash e : B}{\Gamma \vdash \lambda\|x : A\|, e : \forall x : A, B}\,\textsf{Lam-Ir} \qquad \frac{\Gamma \vdash f : \forall x : A, B \quad \Gamma^{\uparrow} \vdash a : A}{\Gamma \vdash f\|a\| : [a/x]B}\,\textsf{App-Ir}$$

$$\frac{\Gamma \vdash e : \exists x : A, B \quad \Gamma, z : \exists x : A, B \vdash C\ \mathsf{ty} \quad \Gamma, \|x : A\|, y : B \vdash e' : [(\|x\|, y)/z]C}{\Gamma \vdash \mathsf{let}\ (\|x\|, y) : \exists x : A, B = e\ \mathsf{in}\ e' : [e/z]C}\,\textsf{Let-Ir}$$

$$\frac{\Gamma, n : \mathbb{N} \vdash C\ \mathsf{ty} \quad \Gamma \vdash e : \mathbb{N} \quad \Gamma \vdash z : [0/n]C \quad \Gamma, \|n : \mathbb{N}\|, y : C \vdash s : [\mathsf{succ}\ n/n]C}{\Gamma \vdash \mathsf{natrec}[n \mapsto C]\ e\ z\ (\|\mathsf{succ}\ x\|, y \mapsto s) : [e/n]C}\,\textsf{Natrec}$$

Fig. 7. λ_{ert} Term Typing

Dependent functions of type $(x : A) \to B$ may be introduced by lambda abstraction, via the rule Lam, and may be eliminated by application, via the rule App. The corresponding β and η equations are introduced axiomatically, with the rules β_{ty} and η_{ty}. Each of the β and η axioms is subscripted with an annotation naming the type former it is for. For example, the subscript "ty" in β_{ty} refers to the fact that dependent function types are parameterized by a term variable (with a type). The rule application itself is annotated with the function body and argument.

While dependent function types abstract over a *computational* variable, we can also abstract over *ghost* or *(computationally) irrelevant* variables, yielding a form of *intersection type*. The type well-formedness conditions for $\forall x : A, B$ (in Intr-WF) are essentially the same conditions as for dependent functions, but the introduction rule Lam-Ir checks the body assuming the parameter is irrelevant.

We can eliminate a term of intersection type by applying it to an expression which is well-typed in the upgraded context $\Gamma^{\uparrow}$, i.e., which may contain ghost variables, via the rule App-Ir. Similarly to the case for dependent functions, reduction must be encoded as an axiom β_{ir}, where the "ir" stands for "irrelevant." We may also introduce an *irrelevance axiom*, Ir-Ty, which essentially says that ghost

$$\frac{}{\Gamma, u : \varphi \vdash u : \varphi}\ \textsf{Var-Pr} \qquad \frac{}{\Gamma \vdash \langle\rangle : \top}\ \textsf{True} \qquad \frac{\Gamma \vdash p : \bot}{\Gamma \vdash \mathsf{absurd}\ p : \varphi}\ \textsf{Absurd-Pr}$$

$$\frac{\Gamma, u : \varphi \vdash p : \psi}{\Gamma \vdash \hat{\lambda} u : \varphi, p : (u : \varphi) \Rightarrow \psi}\ \textsf{Imp} \qquad \frac{\Gamma \vdash p : (u : \varphi) \Rightarrow \psi \quad \Gamma \vdash q : \psi}{\Gamma \vdash p\ q : [q/u]B}\ \textsf{MP}$$

$$\frac{\Gamma \vdash p : \varphi \quad \Gamma \vdash q : [p/u]\psi}{\Gamma \vdash \langle p, q\rangle : (u : \varphi) \wedge \psi}\ \textsf{And} \qquad \frac{\Gamma \vdash p : \varphi}{\Gamma \vdash \mathsf{orl}\ p : \varphi \vee \psi}\ \textsf{Orl} \qquad \frac{\Gamma \vdash p : \psi}{\Gamma \vdash \mathsf{orr}\ p : \varphi \vee \psi}\ \textsf{Orr}$$

$$\frac{\Gamma \vdash p : (u : \varphi) \wedge \psi \quad \Gamma, w : (u : \varphi) \wedge \psi \vdash \theta\ \mathsf{pr} \quad \Gamma, u : \varphi, v : \psi \vdash q : [\langle u, v\rangle / w]\theta}{\Gamma \vdash \mathsf{let}\ \langle u, v\rangle : (u : \varphi) \wedge \psi = p\ \mathsf{in}\ q : [p/w]\theta}\ \textsf{Let-And}$$

$$\frac{\Gamma, u : \varphi \vee \psi \vdash \theta\ \mathsf{pr} \quad \Gamma \vdash p : \varphi \vee \psi \quad \Gamma, v : \varphi \vdash l : [\mathsf{orl}\ v/u]\theta \quad \Gamma, w : \psi \vdash r : [\mathsf{orr}\ w/u]\theta}{\Gamma \vdash \mathsf{cases}_{\mathsf{or}}\ [u \mapsto \theta]\ p\ (\mathsf{orl}\ v \mapsto l)\ (\mathsf{orr}\ w \mapsto r) : [e/u]C}\ \textsf{Cases-Or}$$

$$\frac{\Gamma, \|x : A\| \vdash p : \varphi}{\Gamma \vdash \hat{\lambda}\|x : A\|, p : \forall x : A, \varphi}\ \textsf{Gen} \qquad \frac{\Gamma \vdash p : \forall x : A, \varphi \quad \Gamma^{\uparrow} \vdash a : A}{\Gamma \vdash p\ \|a\| : [a/x]\varphi}\ \textsf{Spec}$$

$$\frac{\Gamma^{\uparrow} \vdash a : A \quad \Gamma \vdash p : [a/x]\varphi}{\Gamma \vdash \langle \|a\|, p\rangle : \exists x : A, \varphi}\ \textsf{Wit}$$

$$\frac{\Gamma \vdash p : \exists x : A, \varphi \quad \Gamma, v : \exists x : A, \varphi \vdash \psi\ \mathsf{pr} \quad \Gamma, x : A, u : \varphi \vdash q : [\langle \|a\|, p\rangle / v]\psi}{\Gamma \vdash \mathsf{let}\ \langle \|x\|, u\rangle : \exists x : A, \varphi = p\ \mathsf{in}\ q : [p/v]\psi}\ \textsf{Let-Exists}$$

$$\frac{\Gamma^{\uparrow} \vdash e : (x : A) \times B \quad \Gamma, z : (x : A) \times B \vdash \varphi\ \mathsf{pr} \quad \Gamma, x : A, y : B \vdash e' : [(x, y)/z]\varphi}{\Gamma \vdash \mathsf{let}\ (x, y) : (x : A) \times B = e\ \mathsf{in}\ e' : [e/z]\varphi}\ \textsf{Let-Pair-Pr}$$

$$\frac{\Gamma^{\uparrow} \vdash e : \{x : A \mid \varphi\} \quad \Gamma, z : \{x : A \mid \varphi\} \vdash \psi\ \mathsf{pr} \quad \Gamma, x : A, u : \varphi \vdash e' : [\{x, y\}/z]\psi}{\Gamma \vdash \mathsf{let}\ \{x, u\} : \{x : A \mid \varphi\} = e\ \mathsf{in}\ e' : [e/z]\psi}\ \textsf{Let-Set-Pr}$$

$$\frac{\Gamma^{\uparrow} \vdash e : \exists x : A, \varphi \quad \Gamma, z : \exists x : A, B \vdash \varphi\ \mathsf{ty} \quad \Gamma, x : A, y : B \vdash e' : [(\|x\|, y)/z]\varphi}{\Gamma \vdash \mathsf{let}\ (\|x\|, y) : \exists x : A, B = e\ \mathsf{in}\ e' : [e/z]\varphi}\ \textsf{Let-Ir-Pr}$$

$$\frac{\Gamma^{\uparrow} \vdash a : A \quad \Gamma^{\uparrow} \vdash b : A \quad \Gamma \vdash p : a =_A b \quad \Gamma \vdash q : [a/x]\varphi}{\Gamma \vdash \mathsf{subst}[x \mapsto \varphi][a][b]\ p\ q : [b/x]\varphi}\ \textsf{Subst}$$

$$\frac{\Gamma, x : A + B \vdash \varphi; \mathsf{pr} \quad \Gamma \vdash e : A + B \quad \Gamma, y : A \vdash l : [\mathsf{inl}\ y/x]\varphi \quad \Gamma, z : B \vdash r : [\mathsf{inr}\ z/x]\varphi}{\Gamma \vdash \mathsf{cases}\ [x \mapsto \varphi]\ e\ (\mathsf{inl}\ y \mapsto l)\ (\mathsf{inr}\ z \mapsto r) : [e/x]\varphi}\ \textsf{Cases-Pr}$$

$$\frac{\Gamma, n : \mathbb{N} \vdash \varphi\ \mathsf{pr} \quad \Gamma^{\uparrow} \vdash e : \mathbb{N} \quad \Gamma \vdash z : [0/n]\varphi \quad \Gamma, n : \mathbb{N}, y : \varphi : s : [\mathsf{succ}\ n/n]\varphi}{\Gamma \vdash \mathsf{ind}[n \mapsto \varphi]\ e\ z\ (\mathsf{succ}\ n, y \mapsto s) : [e/n]\varphi}\ \textsf{Ind}$$

Fig. 8. λ_{ert} Proof Typing

arguments do not matter for the purposes of determining equality whenever the ghost variable does not occur in the result type.

We move on to introduce dependent pair types with the type formation rule Pair-WF. The introduction rule Pair looks a bit like the introduction rule for sigma-types in dependent type theories, with the type of the second component varying according to the first component, and both components computationally relevant.

The elimination form is a let-binding form (in Let-Pair). We may also eliminate into proofs using Let-Pair-Pr; note that, in this case, the expression $e : (a : A) \times B$ may contain ghost variables (as it

$$\frac{\Gamma^{\uparrow} \vdash a : A}{\Gamma \vdash \mathsf{rfl}\ a : a =_A a}\ \text{Rfl} \qquad \frac{\Gamma^{\uparrow} \vdash a : \mathbf{1}}{\Gamma \vdash \mathsf{uniq}\ a : a =_{\mathbf{1}} ()}\ \text{Uniq}$$

$$\frac{\Gamma^{\uparrow} \vdash a : A \quad \Gamma^{\uparrow} \vdash b : B \quad \Gamma \vdash p : \mathsf{inl}\ a =_{A+B} \mathsf{inr}\ b}{\Gamma \vdash \mathsf{discr}\ a\ b\ p : \bot}\ \text{Discr}$$

$$\frac{\Gamma^{\uparrow}, u : \varphi \vdash e : A \quad \Gamma \vdash p : \varphi}{\Gamma \vdash \beta_{\mathsf{pr}}(u \mapsto e)\ p : (\hat{\lambda} u : \varphi.e)\ p =_{[p/u]A} [p/u]e}\ \beta_{\mathsf{pr}}$$

$$\frac{\Gamma^{\uparrow}, x : A \vdash e : B \quad \Gamma^{\uparrow} \vdash a : A}{\Gamma \vdash \beta_{\mathsf{ty}}(x \mapsto e)\ a : (\lambda x : A.e)\ a =_{[a/x]B} [a/x]e}\ \beta_{\mathsf{ty}}$$

$$\frac{\Gamma^{\uparrow}, x : A \vdash e : B \quad \Gamma^{\uparrow} \vdash a : A}{\Gamma \vdash \beta_{\mathsf{ir}}(x \mapsto e)\ a : (\lambda \|x : A\|.e)\ \|a\| =_{[a/x]B} [a/x]e}\ \beta_{\mathsf{ir}}$$

$$\frac{\Gamma^{\uparrow} \vdash a : A \quad \Gamma^{\uparrow}, y : A \vdash l : [\mathsf{inl}\ y/x]C \quad \Gamma^{\uparrow}, z : B \vdash r : [\mathsf{inr}\ z/x]C}{\begin{array}{ll}\Gamma \vdash & \beta_{\mathsf{left}}[x \mapsto C]\ (\mathsf{inl}\ y \mapsto l)\ (\mathsf{inr}\ z \mapsto r)\ (\mathsf{inl}\ a) \\ \quad : & \mathsf{cases}[x \mapsto C]\ (\mathsf{inl}\ a)\ (\mathsf{inl}\ y \mapsto l)\ (\mathsf{inr}\ z \mapsto r) =_{[\mathsf{inl}\ a/x]C} [a/y]l\end{array}}\ \beta_{\mathsf{left}}$$

$$\frac{\Gamma^{\uparrow} \vdash b : B \quad \Gamma^{\uparrow}, y : A \vdash l : [\mathsf{inl}\ y/x]C \quad \Gamma^{\uparrow}, z : B \vdash r : [\mathsf{inr}\ z/x]C}{\begin{array}{ll}\Gamma \vdash & \beta_{\mathsf{right}}[x \mapsto C]\ (\mathsf{inl}\ y \mapsto l)\ (\mathsf{inr}\ z \mapsto r)\ (\mathsf{inr}\ b) \\ \quad : & \mathsf{cases}[x \mapsto C]\ (\mathsf{inr}\ b)\ (\mathsf{inl}\ y \mapsto l)\ (\mathsf{inr}\ z \mapsto r) =_{[\mathsf{inr}\ b/x]C} [b/z]r\end{array}}\ \beta_{\mathsf{right}}$$

$$\frac{\Gamma^{\uparrow} \vdash z : [0/x]C \quad \Gamma^{\uparrow}, x : \mathbb{N}, y : C \vdash s : [\mathsf{succ}\ x/x]C}{\Gamma \vdash \beta_{\mathsf{zero}}[x \mapsto C]\ z\ (\|\mathsf{succ}\ x\|, y \mapsto s) : \mathsf{natrec}[x \mapsto C]\ 0\ z\ (\|\mathsf{succ}\ x\|, y \mapsto s) =_{[0/x]C} z}\ \beta_{\mathsf{zero}}$$

$$\frac{\Gamma^{\uparrow} \vdash e : \mathbb{N} \quad \Gamma^{\uparrow} \vdash z : [0/x]C \quad \Gamma^{\uparrow}, x : \mathbb{N}, y : C \vdash s : [\mathsf{succ}\ x/x]C}{\begin{array}{ll}\Gamma \vdash & \beta_{\mathsf{succ}}[x \mapsto C]\ (\mathsf{succ}\ e)\ z\ (\|\mathsf{succ}\ x\|, y \mapsto s) \\ \quad : & \mathsf{natrec}[x \mapsto C]\ (\mathsf{succ}\ e)\ z\ (\|\mathsf{succ}\ x\|, y \mapsto s) \\ & =_{[\mathsf{succ}\ e/x]C} [(\mathsf{natrec}[x \mapsto C]\ e\ z\ (\|\mathsf{succ}\ x\|, y \mapsto s))/y][e/x]s\end{array}}\ \beta_{\mathsf{succ}}$$

$$\frac{\Gamma^{\uparrow} \vdash a : A \quad \Gamma^{\uparrow} \vdash b : [a/y]B \quad \Gamma^{\uparrow}, y : A, z : B \vdash e : [(y, z)/x]C}{\Gamma \vdash \beta_{\mathsf{pair}}\ (a, b)\ ((y, z) \mapsto e) : (\mathsf{let}\ (y, z) = (a, b)\ \mathsf{in}\ e) =_{[(a,b)/x]C} [b/z][a/y]e}\ \beta_{\mathsf{pair}}$$

$$\frac{\Gamma^{\uparrow} \vdash a : A \quad \Gamma^{\uparrow} \vdash p : [a/y]\varphi \quad \Gamma^{\uparrow}, y : A, u : \varphi \vdash e : [\{y, u\}/x]C}{\Gamma \vdash \beta_{\mathsf{set}}\ \{a, p\}\ ((y, u) \mapsto e) : (\mathsf{let}\ \{y, u\} = \{a, p\}\ \mathsf{in}\ e) =_{[\{a,p\}/x]C} [p/u][a/y]e}\ \beta_{\mathsf{set}}$$

$$\frac{\Gamma^{\uparrow} \vdash a : A \quad \Gamma^{\uparrow} \vdash b : [a/y]B \quad \Gamma^{\uparrow}, y : A, z : B \vdash e : [(\|y\|, z)/x]C}{\Gamma \vdash \beta_{\mathsf{repr}}\ (\|a\|, b)\ ((y, z) \mapsto e) : (\mathsf{let}\ (\|y\|, z) = (\|a\|, b)\ \mathsf{in}\ e) =_{[(\|a\|,b)/x]C} [b/z][a/y]e}\ \beta_{\mathsf{repr}}$$

$$\frac{\Gamma^{\uparrow} \vdash f : (x : A) \to B}{\Gamma \vdash \eta_{\mathsf{ty}}\ f : \lambda x : A, f\ x =_{(x:A)\to B} f}\ \eta_{\mathsf{ty}} \qquad \frac{\Gamma \vdash A\ \mathsf{ty} \quad \Gamma^{\uparrow}, u : \varphi \vdash e : A \quad \Gamma^{\uparrow} \vdash p : \varphi \quad \Gamma^{\uparrow} \vdash q : \varphi}{\Gamma \vdash \mathsf{ir}_{\mathsf{pr}}[u \mapsto e]\ p\ q : [p/u]e =_A [q/u]e}\ \text{Ir-Pr}$$

$$\frac{\Gamma \vdash B\ \mathsf{ty} \quad \Gamma^{\uparrow} \vdash e : \forall x : A, B \quad \Gamma^{\uparrow} \vdash a : A \quad \Gamma^{\uparrow} \vdash b : A}{\Gamma \vdash \mathsf{ir}_{\mathsf{ty}}\ e\ a\ b : e\ \|a\| =_B e\ \|b\|}\ \text{Ir-Ty}$$

$$\frac{\Gamma^{\uparrow} \vdash f : \forall x : A, B \quad \Gamma^{\uparrow} \vdash g : \forall x : A, B \quad \Gamma^{\uparrow} \vdash i : A \quad \Gamma, x : A \vdash p : f\ \|x\| =_B g\ \|x\|}{\Gamma \vdash \eta_{\mathsf{ir}}\ f\ g\ i\ p : f =_{\forall x:A,B} g}\ \eta_{\mathsf{ir}}$$

$$\frac{\Gamma^{\uparrow} \vdash f : (x : \varphi) \Rightarrow B \quad \Gamma^{\uparrow} \vdash g : (x : \varphi) \Rightarrow B \quad \Gamma^{\uparrow} \vdash i : \varphi \quad \Gamma, x : \varphi \vdash p : f\ x =_B g\ x}{\Gamma \vdash \eta_{\mathsf{pr}}\ f\ g\ i\ p : f =_{(x:\varphi)\Rightarrow B} g}\ \eta_{\mathsf{pr}}$$

Fig. 9. λ_{ert} Axiom Typing only needs to be well-typed in $\Gamma^{\uparrow}$). As before, reduction for dependent pair elimination must be encoded as an axiom, β_{pair}.

Often, however, we may want to be able to consider, dually to intersection types, *union types* $\exists x : A, B(x)$ (Union-WF), which we may views as dependent pairs conditioned on a ghost variable, or, set-theoretically, as elements of $B(a)$ for some valid a. Similarly to for dependent pairs, we support an introduction rule Pair-Ir, and elimination via let-binding using rules Let-Ir and Let-Ir-Pr. Note that, unlike for dependent pairs, and similarly to intersection types, a only needs to be well-typed in $\Gamma^{\uparrow}$ (rather than Γ) in the introduction rule Pair-Ir, while in Let-Ir, the binder $\|x : A\|$ is a ghost binder rather than a term binder. Finally, as before, we also introduce a reduction rule for let-bindings, β_{ir}.

Just as dependent functions and pairs have corresponding type formers quantifying over ghost variables rather than term variables, we may also construct type formers predicated over propositions. In particular, we may consider the *precondition type*: essentially a closure yielding an element of the type A if the proposition p is true; this has introduction rule Pre-WF. Introduction is by abstracting a term over a proof variable, with rule Lam-Pr. Note that the type A in Lam-Pr is allowed to depend on a proof $u : \varphi$; this is because we may consider precondition types $(u : \varphi) \Rightarrow A$ in which A is only well-formed if φ holds. For example, consider a function $f : \{X \mid \varphi(x)\} \rightarrow X$ (we will cover subset types shortly); to reason about values of f for $a : X$, we need $\varphi(a)$ to hold, hence, we require dependency on proofs to be able to type $(u : \varphi(a)) \Rightarrow \{z : \{y : X \mid \varphi(y)\} \mid f\ z = f\ \{a, u\}\}$. Similarly to for the computational and ghost variable cases, we must also introduce a reduction rule β_{pr}. Dually, we may introduce the *subset type* former Set-WF, representing, in essence, elements a of type A satisfying the predicate $\varphi(a)$. This has the expected introduction rule Set and supports elimination via let-binding with rules Let-Set and Let-Set-Pr. We also introduce a reduction rule, β_{set}, as expected.

Currently, we have the ability to manipulate data and associate it with propositions, but do not yet have any bona fide data types. While there is no semantic obstacle to introducing a full language of datatype declarations, for simplicity, we will restrict ourselves to the unit type, sum types, and the natural numbers. Other inductive types follow a similar pattern. As usual, a value of the unit type may be introduced with rule Unit; rather than the eliminator we would expect from dependent type theory, we may simply couple this with an axiom, Uniq, stating that every member of the unit type is equal to (). While on it's own this is not a particularly interesting datatype, when combined with coproduct types $A + B$, we may define, for example, the type of Booleans as $\mathbf{2} \equiv \mathbf{1} + \mathbf{1}$, allowing us to construct finite types. Coproducts may be introduced via injection (via rules Inl and Inr) and eliminated by case splitting (via rules Cases and Cases-Pr). To demonstrate support for infinite types, we introduce the natural numbers $\mathbb{N}$, constants of which may be built up using Zero and Succ. The elimination rule, Natrec, implements essentially iteration with the current step available as a *ghost* variable (for reasoning about in propositions); it's computational semantics are given by the axioms β_{zero} and β_{succ}. Furthermore, just as we may construct terms recursively via natrec, we may also perform proofs by induction via ind using rule Ind. Note that, unlike in Natrec, the expression n over which we are performing induction is allowed to contain ghost variables.

4.2.3 Propositional Structure. Now that we have given essentially a complete description of the λ_{ert}'s terms and their computational behaviour, we can describe the main components of λ_{ert}'s propositional logic, which for the most part mirror the term formers, since we encode proofs in first-order logic as λ-terms via the Curry-Howard correspondence. We begin by introducing propositions $\top$ and $\bot$; the former is only equipped with introduction rule True, whereas $\bot$, being an initial object, is only equipped with the elimination rules Absurd-Pr and Absurd. The latter rule is especially important, as it is the main way with which our logic is capable of interacting with our term calculus by allowing us to safely erase unreachable branches from, e.g., a natrec or cases expression.

Proofs of equality do not interact meaningfully with the term calculus, since they are logical formulae, and our logic is classical and nonconstructive. So principles such as unique choice (which permit turning a proof that there exists a unique element of type A into a computational value of type A) are not valid. The only case in which this is allowable is for the empty type (see the discussion of the Absurd rule above). To make use of this, we need the additional axiom Discr, which essentially says that the right-hand and left-hand side of a coproduct type are disjoint. This suffices to effectively introduce disequality into our type theory, as desired.

We may now introduce the rest of the connectives of first-order logic, suitably modified in order to fit our setting. In particular, we have *implication* $(u : \varphi) \Rightarrow \psi$, which, as per the Curry-Howard correspondence, is introduced by abstracting over a proposition variable, via rule Imp, and eliminated via modus ponens (under Curry-Howard, application) MP. Similarly, we have *conjunction* $(u : \varphi) \wedge \psi$, which is introduced by constructing a pair, via rule And, and eliminated via a let-binding, with rule Let-And. We note that both associated proposition formers, Imp-WF and And-WF, are "dependent," in that their right-hand side ψ is allowed to depend on a proof variable u for the left-hand side φ. This is because we allow the case where ψ is not well-formed without φ holding (for example, because it is about a term that requires a proof of φ). On the other hand, similarly to for coproducts, the rules for disjunction $\varphi \vee \psi$ are simpler, with introduction by injection via rules OrI and Orr, and elimination via case splitting with rule Cases-Or.

Finally, just as we may consider a type quantifying over a proposition, to support full first-order logic, we must also be able to consider propositions quantified over types. In particular, we may introduce *universally quantified* propositions with formation rule Univ-Wf. These may be introduced by generalization via rule Gen, and specialized via elimination rule Spec. Similarly, we may introduce *existentially quantified* propositions with formation rule Exists-WF. We introduce proofs of an existentially quantified proposition by introducing a witness via rule Wit, and may eliminate proofs via let-binding with rule Let-Exists. Note in particular that, in both cases, we treat the variable being quantified over as a *ghost* variable, since it is appearing in a proposition. Furthermore, since propositions have no computational semantics, a reduction rule is unnecessary for either.

Altogether, our β and η rules are quite powerful, enabling us to prove more extensionality properties than a casual look at the axioms may suggest. In particular, fixing a term e with free variable $z : \{x : A \mid \varphi\}$, the η-rule for subsets can be written as:

$$\begin{aligned} \eta_{\text{set}}[C] \quad & \equiv \hat{\lambda}\|a : \{x : A \mid \varphi\}\|, \mathsf{let}\ \{y, v\} = a\ \mathsf{in}\ \beta_{\text{set}}\ \{y, v\}\ ((x, u) \mapsto ([\{x, u\}/z]e)) \\ & : \forall a : \{x : A \mid \varphi\}, \mathsf{let}\ \{x, u\} = a\ \mathsf{in}\ [\{x, u\}/z]e = [a/z]e \end{aligned} \tag{19}$$

(note that a, x, u, y, v are assumed to be fresh variables here). In general, we only need to introduce explicit extensionality axioms for intersection types (η_{ir}) and precondition types (η_{pr}), with all the other η and extensionality rules we would expect to hold being derivable from the rest of the axioms, with the sole exception of function extensionality, which is not compatible with our current semantics. See section 5.3 for the explanation of why.

Overall, λ_{ert}'s logic is essentially multi-sorted first-order logic, with the sorts drawn from the types of λ_{ert}'s programming language. Since λ_{ert} has function types, this means that the λ_{ert} logic is fairly close to PA$^{\omega}$, Peano arithmetic over the full type hierarchy. So any property provable about λ_{ert} terms with first-order logic and induction should be provable. (We have not proved any theorems about the expressivity of λ_{ert}'s logic, though.)

4.3 Syntactic Metatheory

λ_{ert} satisfies the expected syntactic properties of substitution and regularity. To show this, we define substitutions as functions $\sigma : \mathsf{Var} \to \mathsf{Term} \uplus \mathsf{Proof}$, and can recursively define capture-avoiding substitution on terms/proofs e, written $[\sigma]e$, in the obvious way. We define a well-formed substitution from Γ to Δ, written $\Gamma \vdash' \sigma : \Delta$, as a substitution satisfying the following conditions:

$$\begin{aligned}
[(x : A) \in \Gamma] &\implies \Delta \vdash \sigma\, x : [\sigma]A \\
[(u : \varphi) \in \Gamma] &\implies \Delta \vdash \sigma\, u : [\sigma]\varphi \\
[\|x : A\| \in \Gamma] &\implies [\|\sigma\, x : [\sigma]A\| \in \Delta] \vee [\Delta \vdash \sigma\, x : [\sigma]A]
\end{aligned} \tag{20}$$

Furthermore, we say σ is a *strict* substitution, written $\Gamma \vdash \sigma : \Delta$, if ghost variables in Γ are only replaced with ghost variables in Δ.

Lemma 4.3 (Syntactic Substitution). *If* $\Gamma \vdash' \sigma : \Delta$ *and* $\Gamma \vdash a : A$, *then* $\Delta \vdash [\sigma]a : [\sigma]A$.
Formalized as: `LogicalRefinement/Typed/Subst.lean`, `theorem HasType.subst'`

The proof of substitution is a routine induction, which as usual requires first proving weakening. Once we know that substitution holds, we can prove regularity:

Lemma 4.4 (Syntactic Regularity). *If* $\Gamma \vdash a : A$, *then* $\Gamma \vdash A$ ty. *Also, if* $\Gamma \vdash p : \varphi$, *then* $\Gamma \vdash \varphi$ pr.
Formalized as: `LogicalRefinement/Typed/Regular.lean`, `theorem HasType.regular`

This requires syntactic substitution since some of the typing rules (such as App) involve a substitution in the result types. One other result we will use later is that substitutions can be upgraded; that is, $\Gamma \vdash \sigma : \Delta \implies \Gamma^{\uparrow} \vdash \sigma : \Delta^{\uparrow}$. To avoid confusion, we write the latter as $\Gamma^{\uparrow} \vdash \sigma^{\uparrow} : \Delta^{\uparrow}$, with the upgrade on substitutions taken to be the identity.

5 SEMANTICS

To give a denotational semantics for the λ_{ert} calculus, we first show that all the proofs and dependencies in an λ_{ert} term can be erased in a compositional way. This yields a simple type for each λ_{ert} type, and a simply-typed term for each λ_{ert} term. We then give a semantics for each λ_{ert} type as a subset of the denotational semantics of the erasure for that type. Finally, we show that each well-typed λ_{ert} term lies in the subset defined by its type.

In section 5.1, we recall the syntax and semantics of the simply-typed lambda calculus. In section 5.2, we give an erasure function from λ_{ert} types and terms to λ_{stlc} types and terms respectively, and prove some expected properties like preservation of well-typedness and semantic substitution (where the denotational semantics of an λ_{ert} term are taken to be the semantics of its erasure). Finally, in section 5.3, we give a semantics to λ_{ert} types by assigning each a subset, and show that the denotations of all well-typed λ_{ert} terms lie in the subset assigned to their type, i.e., semantic regularity. From this, we deduce that "well typed programs don't go wrong".

For the more categorically-minded reader, we interpret λ_{stlc} in the Kleisli category of the exception monad $M(X)$ (in the category of sets), and then interpret λ_{ert} in terms of the subset fibration over it. That is, an λ_{ert} type can be understood as a pair $(X, P \subseteq X)$, where X is a type of λ_{stlc}, and P is a predicate on that type; and λ_{ert} terms are maps $f \in (\Gamma, P_\Gamma) \to (M(X), P_{M(X)})$ such that for all $\gamma \in P_\Gamma$, we have $f(\gamma) \in P_{M(X)}$. Semantically, our erasure operation amounts to the forgetful functor into the Kleisli category which drops the property information.

5.1 The Simply-Typed Lambda Calculus

We begin by providing typing rules for λ_{stlc} in figure 10 (a formal grammar is provided in the appendix). This is a standard lambda calculus with functions, sums, products, natural numbers, as

$$\frac{}{\Gamma \vdash_\lambda () : \mathbf{1}} \qquad \frac{}{\Gamma \vdash_\lambda \mathsf{error} : A} \qquad \frac{\Gamma \vdash_\lambda a : A \quad \Gamma, x : A \vdash_\lambda e : B}{\Gamma \vdash_\lambda \mathsf{let}\ x = a\ \mathsf{in}\ e : B}$$

$$\frac{\Gamma, x : A \vdash_\lambda e : B}{\Gamma \vdash_\lambda \lambda x : A, e : A \to B} \qquad \frac{\Gamma \vdash_\lambda f : A \to B \quad \Gamma \vdash_\lambda a : A}{\Gamma \vdash_\lambda f\ a : B} \qquad \frac{\Gamma \vdash_\lambda l : A \quad \Gamma \vdash_\lambda r : B}{\Gamma \vdash_\lambda (l, r) : A \times B}$$

$$\frac{\Gamma \vdash_\lambda e : A}{\Gamma \vdash_\lambda \mathsf{inl}\ e : A + B} \qquad \frac{\Gamma \vdash_\lambda e : B}{\Gamma \vdash_\lambda \mathsf{inr}\ e : A + B} \qquad \frac{\Gamma \vdash_\lambda e : A + B \quad \Gamma, x : A \vdash_\lambda l : C \quad \Gamma, y : B \vdash_\lambda r : C}{\Gamma \vdash_\lambda \mathsf{cases}\ e\ (\mathsf{inl}\ x \mapsto l)\ (\mathsf{inr}\ y \mapsto r) : C}$$

$$\frac{}{\Gamma \vdash_\lambda 0 : \mathbb{N}} \qquad \frac{}{\Gamma \vdash_\lambda \mathsf{succ} : \mathbb{N} \to \mathbb{N}} \qquad \frac{\Gamma \vdash_\lambda e : \mathbb{N} \quad \Gamma \vdash_\lambda z : C \quad \Gamma, x : C \vdash_\lambda s : C}{\Gamma \vdash_\lambda \mathsf{natrec}\ e\ z\ (x \mapsto s) : C}$$

Fig. 10. λ_{stlc} typing rules

well as a simple effect: error stops. We define Term_λ to be the set of λ_{stlc} terms. Similarly to the λ_{ert} calculus, given a function $\sigma : \mathsf{Var} \to \mathsf{Term}_\lambda$, we may then recursively define (capture-avoiding) substitution of a term t in the usual manner. We say that σ is a substitution from Γ to Δ, written $\Gamma \vdash_\lambda \sigma : \Delta$, if it satisfies the property that $(a : A) \in \Gamma \implies \Delta \vdash_\lambda \sigma\ a : A$.

We may then state the usual property of syntactic substitution as follows:

Lemma 5.1 (Syntactic Substitution (λ_{stlc})). *Given a substitution* $\Gamma \vdash_\lambda \sigma : \Delta$ *and* $\Gamma \vdash_\lambda t : A$*, we have* $\Delta \vdash_\lambda [\sigma]t : A$

Formalized as: `LogicalRefinement/Stlc/Basic.lean`, `theorem Stlc.HasType.subst`

We may now give λ_{stlc} a denotational semantics. Fixing the exception monad M, with exception error, we begin by giving denotations for λ_{stlc} types in figure 11, using Moggi's call-by-value semantics for types [Moggi 1991]. We may then define the denotation of an λ_{stlc} context elementwise by taking $[\![\cdot]\!] = \mathbf{1}$, $[\![\Gamma, x : A]\!] = [\![\Gamma]\!] \times \mathsf{M}\ [\![A]\!]$. Despite the fact that our semantics is call-by-value, the interpretation of each hypothesis lives in the monad M. We do this so our denotational semantics can interpret substituting arbitrary terms for variables, and not just values for variables. (Since the substitution rule permits substituting arbitrary terms for variables, our semantics has to support this too.) For each variable $x : A$ in a context Γ, we define pointwise projections $\pi_x : [\![\Gamma]\!] \to [\![A]\!]$. We define (in figure 12) the denotation of a derivation $\Gamma \vdash_\lambda a : A$ as a function of type $[\![\Gamma]\!] \to \mathsf{M}([\![A]\!])$, which takes environments (elements of $[\![\Gamma]\!]$) to elements of the monadic type $\mathsf{M}([\![A]\!])$.

We can now give a relatively straightforward account of the denotational semantics of an λ_{stlc} substitution as follows: we interpret a substitution $\Gamma \vdash_\lambda \sigma : \Delta$ as a function $[\![\Gamma \vdash_\lambda \sigma : \Delta]\!] : [\![\Delta]\!] \to [\![\Gamma]\!]$ by composing it elementwise with the the denotational semantics. Supposing $D \in [\![\Delta]\!]$:

$$\pi_{x,\Gamma}([\![\Gamma \vdash_\lambda \sigma : \Delta]\!]\ D) = [\![\Delta \vdash_\lambda \sigma\ x : A]\!]\ D \in [\![A]\!] \tag{21}$$

We may now state semantic substitution for the λ_{stlc} as follows:

Lemma 5.2 (Semantic Substitution (λ_{stlc})). *Given* λ_{stlc} *derivation* $\Gamma \vdash_\lambda a : A$ *and* λ_{stlc} *substitution* $\Gamma \vdash_\lambda \sigma : \Delta$*, we have*

$$[\![\Gamma \vdash_\lambda a : A]\!] \circ [\![\Gamma \vdash_\lambda \sigma : \Delta]\!] = [\![\Delta \vdash_\lambda [\sigma]a : [\sigma]A]\!]$$

Formalized as: `LogicalRefinement/Stlc/Subst.lean`, `theorem Stlc.HasType.subst_interp_dist`

5.2 Erasure

We define a notion of erasure $|A|$ of λ_{ert} types and terms to corresponding λ_{stlc} ones in figure 13. Erasure on types simply erases all dependency and propositional information leaving behind a

$$
\llbracket \mathbf{0} \rrbracket = \{\}, \qquad \llbracket \mathbf{1} \rrbracket = \{*\}, \qquad \llbracket \mathbb{N} \rrbracket = \mathbb{N}
$$
$$
\llbracket A \to B \rrbracket = \llbracket A \rrbracket \to \mathsf{M}\llbracket B \rrbracket, \qquad \llbracket A + B \rrbracket = \llbracket A \rrbracket \sqcup \llbracket B \rrbracket, \qquad \llbracket A \times B \rrbracket = \llbracket A \rrbracket \times \llbracket B \rrbracket
$$

Fig. 11. λ_{stlc} type denotations parameterized by a monad M. The denotation of a term of type A has type $\mathsf{M}\llbracket A \rrbracket$.

$$
\boxed{\llbracket \Gamma \vdash_\lambda a : A \rrbracket : \llbracket \Gamma \rrbracket \to \mathsf{M}\ \llbracket A \rrbracket}
$$

$$
\begin{aligned}
\llbracket \Gamma \vdash_\lambda x : A \rrbracket\ G &= \pi_{x,\Gamma} G \\
\llbracket \Gamma \vdash_\lambda () : \mathbf{1} \rrbracket\ G &= \mathsf{ret}\ () \\
\llbracket \Gamma \vdash_\lambda \mathsf{error} : A \rrbracket\ G &= \mathsf{error}_A \\
\llbracket \Gamma \vdash_\lambda \lambda x : A, e : A \to B \rrbracket\ G &= \mathsf{ret}\ (\lambda a, \llbracket \Gamma, x : A \vdash_\lambda e : B \rrbracket (G, \mathsf{ret}\ a)) \\
\llbracket \Gamma \vdash_\lambda f\ a : B \rrbracket\ G &= \mathsf{bind}\ (\llbracket \Gamma \vdash_\lambda f : (x : A) \to B \rrbracket\ G)\ (\lambda f, \mathsf{bind}\ (\llbracket \Gamma \vdash_\lambda a : A \rrbracket\ G)\ f) \\
\llbracket \Gamma \vdash_\lambda (l, r) : A \times B \rrbracket\ G &= \mathsf{bind}\ (\llbracket \Gamma \vdash_\lambda l : A \rrbracket\ G)\ (\lambda l, \mathsf{bind}\ (\llbracket \Gamma \vdash_\lambda r : B \rrbracket\ G)\ (\lambda r, \mathsf{ret}\ (l, r))) \\
\llbracket \Gamma \vdash_\lambda \mathsf{inl}\ e : A + B \rrbracket\ G &= \mathsf{fmap}\ \mathsf{inl}\ (\llbracket \Gamma \vdash_\lambda e : A \rrbracket\ G) \\
\llbracket \Gamma \vdash_\lambda \mathsf{inr}\ e : A + B \rrbracket\ G &= \mathsf{fmap}\ \mathsf{inr}\ (\llbracket \Gamma \vdash_\lambda e : B \rrbracket\ G) \\
\left\llbracket \begin{array}{l} \Gamma \vdash_\lambda \mathsf{cases}\ d \\ \qquad (\mathsf{inl}\ y \mapsto l) \\ \qquad (\mathsf{inr}\ z \mapsto r) : C \end{array} \right\rrbracket G &= \left(\begin{array}{l} \mathsf{bind}\ (\llbracket \Gamma \vdash_\lambda d : A + B \rrbracket\ G)\ (\lambda d, \\ \quad \mathsf{cases}\ d \\ \qquad (\mathsf{inl}\ a \mapsto \llbracket \Gamma, y : A \vdash_\lambda l : C \rrbracket\ (G, \mathsf{ret}\ a)) \\ \qquad (\mathsf{inr}\ b \mapsto \llbracket \Gamma, z : B \vdash_\lambda r : C \rrbracket\ (G, \mathsf{ret}\ b)) \end{array} \right) \\
\llbracket 0 \rrbracket\ G &= \mathsf{ret}\ 0 \\
\llbracket \mathsf{succ} \rrbracket\ G &= \mathsf{ret}\ \mathsf{succ} \\
\llbracket \Gamma \vdash_\lambda \mathsf{natrec}\ n\ z\ (x \mapsto s) : C \rrbracket\ G &= \left(\begin{array}{l} \mathsf{bind}\ (\llbracket \Gamma \vdash_\lambda n : \mathbb{N} \rrbracket\ G)\ (\lambda n, \\ \quad \mathsf{natrec}\ n\ (\llbracket \Gamma \vdash_\lambda z : C \rrbracket\ G) \\ \qquad (c \mapsto \mathsf{bind}\ c\ (\lambda c. \llbracket \Gamma, x : C \vdash s : C \rrbracket (G, \mathsf{ret}\ c))) \end{array} \right) \\
\llbracket \Gamma \vdash_\lambda \mathsf{let}\ x = a\ \mathsf{in}\ e : B \rrbracket\ G &= \mathsf{bind}\ (\llbracket \Gamma \vdash_\lambda a : A \rrbracket\ G)\ (\lambda a', \llbracket \Gamma, x : A \vdash_\lambda e : B \rrbracket (G, \mathsf{ret}\ a'))
\end{aligned}
$$

Fig. 12. Denotations for λ_{stlc} terms, where M is the exception monad with $\mathsf{error}_A : \mathsf{M}\ A$

simply-typed skeleton. For tuple-like type formers like $\{x : A \mid \varphi\}$, the propositional information is erased completely, yielding $|A|$, whereas for function-like type formers like $(u : \varphi) \Rightarrow A$, it is instead erased to a unit, yielding $\mathbf{1} \to |A|$; this is to avoid issues with eager evaluation. Where necessary, we take proofs and propositions to erase into the unit as a convenience, i.e.,

$$
\forall \varphi \in \mathsf{Prop}, |\varphi| = \mathbf{1} \in \mathsf{Type}_\lambda, \qquad \forall p \in \mathsf{Proof}, |p| = () \in \mathsf{Term}_\lambda \tag{22}
$$

We may then recursively define the erasure of an λ_{ert} context into an λ_{stlc} context as follows:

$$
|\cdot| = \cdot, \qquad |\Gamma, x : A| = |\Gamma|, x : |A| \qquad |\Gamma, u : \varphi| = |\Gamma|, u : \mathbf{1} \qquad |\Gamma, \|x : A\|| = |\Gamma|, x : \mathbf{1} \tag{23}
$$

As one would expect; erasing a well-typed λ_{ert} term yields a well-typed λ_{stlc} term; in particular, we have that:

Lemma 5.3 (Erasure). *Given a derivation $\Gamma \vdash a : A$, we may derive a derivation of $|\Gamma| \vdash_\lambda |a| : |A|$, written $|\Gamma \vdash a : A|$*

$$\boxed{|\cdot| : \mathsf{Type} \to \mathsf{Type}_\lambda}$$

$$|(x : A) \to B| = |A| \to |B|, \qquad |(x : A) \times B| = |A| \times |B|$$
$$|(u : \varphi) \Rightarrow A| = \mathbf{1} \to |A|, \qquad |\{x : A \mid \varphi\}| = |A|, \qquad |\forall x : A, B| = \mathbf{1} \to |B|, \qquad |\exists x : A, B| = |B|$$
$$|\mathbf{1}| = \mathbf{1}, \qquad |A + B| = |A| + |B|, \qquad |\mathbb{N}| = \mathbb{N}$$

$$\boxed{|\cdot| : \mathsf{Term} \to \mathsf{Term}_\lambda}$$

$$|x| = x \qquad |\lambda x : A, e| = \lambda x : |A|.|e| \qquad |a\ b| = |a|\ |b|, \qquad |(a, b)| = (|a|, |b|)$$
$$|\mathsf{let}\ (x, y) : A = e\ \mathsf{in}\ e'| = \mathsf{let}\ (x, y) = |e|\ \mathsf{in}\ |e'|, \qquad |\mathsf{inl}\ e| = \mathsf{inl}\ |e|, \qquad |\mathsf{inr}\ e| = \mathsf{inr}\ |e|$$
$$|\mathsf{cases}[x \mapsto C]\ e\ (\mathsf{inl}\ y \mapsto l)\ (\mathsf{inr}\ z \mapsto r)| = |\mathsf{cases}\ e\ (\mathsf{inl}\ y \mapsto |l|)\ (\mathsf{inr}\ z \mapsto |r|)|$$
$$|\lambda u : \varphi, e| = \lambda\_ : \mathbf{1}.|e|, \qquad |a\ p| = |a|\ (), \qquad |\{a, p\}| = |a|$$
$$|\mathsf{let}\ \{x, y\} : A = e\ \mathsf{in}\ e'| = \mathsf{let}\ x = |e|\ \mathsf{in}\ |e'|$$
$$|\lambda \|x : A\|, e| = \lambda\_ : \mathbf{1}.|e|, \qquad |a\ \|b\|| = |a|\ (), \qquad |(\|a\|, b)| = |b|$$
$$|\mathsf{let}\ (\|x\|, y) : A = e\ \mathsf{in}\ e'| = \mathsf{let}\ y = |e|\ \mathsf{in}\ |e'|$$
$$|0| = 0 \qquad |\mathsf{succ}| = \mathsf{succ} \qquad |\mathsf{natrec}[x \mapsto C]\ e\ z\ (\|\mathsf{succ}\ n\|, y \mapsto b)| = \mathsf{natrec}\ |e|\ |z|\ (y \mapsto |b|)$$
$$|\mathsf{absurd}\ p| = \mathsf{error}$$

Fig. 13. Erasure of λ_{ert} to λ_{stlc}

Formalized as: `LogicalRefinement/Stlc/Interp.lean`, theorem `HasType.stlc`

With this definition in hand, we may define the erasure of a substitution pointwise, that is, as

$$\forall \sigma \in \mathsf{Var} \to \mathsf{Term}, |\sigma|\ a = |\sigma\ a| \tag{24}$$

It then follows as a trivial corollary of lemma 5.3 that, given a substitution $\Gamma \vdash \sigma : \Delta$, we have $|\Gamma| \vdash_\lambda |\sigma| : |\Delta|$; we write this as $|\Gamma \vdash \sigma : \Delta|$. We are now in a position to prove that the erasure of the substitution of an λ_{ert} term is the erased substitution of the corresponding erased λ_{stlc} term:

Lemma 5.4 (Substitution and Erasure Commute). *Given a substitution $\Gamma \vdash \sigma : \Delta$, we have $|[\sigma]a| = [|\sigma|]|a|$*

Formalized as: `LogicalRefinement/Stlc/InterpSubst.lean`, theorem `HasType.subst_stlc_commute`

We may then deduce the following corollary from lemma 5.2 and lemma 5.4:

Corollary 5.5 (Substitution and Erasure Denotation Commute). *Given λ_{ert} derivation $\Gamma \vdash a : A$ and λ_{ert} substitution $\Gamma \vdash \sigma : \Delta$, we have*

$$[\![|\Gamma \vdash a : A|]\!] \circ [\![|\Gamma \vdash \sigma : \Delta|]\!] = [\![|\Delta \vdash [\sigma]a : [\sigma]A|]\!]$$

Formalized as: `LogicalRefinement/Stlc/InterpSubst.lean`, theorem `HasType.subst_stlc_commute`

5.3 Denotational Semantics

Using lemma 5.3, we could assign a computational meaning to well-typed λ_{ert} terms $\Gamma \vdash t : A$ by simply composing the denotation for λ_{stlc} terms with the erasure function, i.e., taking $[\![|\Gamma \vdash t : A|]\!]$:

$[\![|\Gamma|]\!] \to \mathsf{M}[\![|A|]\!]$. While this interpretation assigns terms a computational meaning, it simply ignores their refinements: there is not yet any guarantee that the refinements mean anything.

To rectify this, we will give a denotational semantics for λ_{ert} types (in figure 14), which maps each type to a subset of the denotation of the corresponding erased type. This semantics is mutually recursive with semantics for λ_{ert} propositions as well.

The denotation of types (and propositions) is parameterized by an environment G drawn from the interpretation of the context Γ. To break the recursion between the semantics of contexts and types, the domain of the interpretation function for types (and propositions) is not restricted to valid λ_{ert} environments $G \in [\![\Gamma \text{ ok}]\!]$, but is defined for all λ_{stlc} environments $G \in [\![|\Gamma|]\!]$. However, the semantics of λ_{ert} types does depend upon the values of ghost variables. As a result, we do not consider the erasure $|\Gamma|$, but rather the erasure of the *upgrade* $|\Gamma^{\uparrow}|$.

The denotation of types basically follows the structure of a unary logical relation (i.e., a logical predicate). For example, the intepretation $[\![\Gamma \vdash (x : A) \to B]\!]\ G$ are functions $|A| \to \mathsf{M}(|B|)$ which satisfy the property that for all inputs in the refined type A, the function returns a pure value in the refined type B. An element of a pair $(x : A) \times B[x]$ is a pair (a, b) in $|A| \times |B|$ where a is an element of A and b is an element of $B[a]$.

The union type $\exists x : A, B[x]$ are those elements of $|B|$ which lie in $B[a]$ for some a in A. The intersection type $\forall x : A, B$ is almost dual, but to account for call-by-value evaluation in the case where A may be an empty type, we consider thunks $1 \to \mathsf{M}(|B|)$ rather than elements of $|B|$. This is also where the terms "intersection type" and "union type" come from – the semantics of $\forall x : A, B$ is literally a giant intersection, and likewise $\exists x : A, B[x]$ is interpreted with a giant union.

An element of a subset type $\{x : A \mid \varphi\}$ is an element of A which also satisfies the property φ. The dual precondition type $(u : \varphi) \Rightarrow A$ represents elements of A, conditional on ϕ holding. Just as with intersections, this type must be represented by thunks $1 \to \mathsf{M}(|A|)$ to account for the case where φ is false. Units and natural numbers have the same denotation as their simply-typed counterpart, and a coproduct $A + B$ is either a left injection of a value satisfying A or a right injection of a value satisfying B. A proposition φ could be interpreted by a map $[\![|\Gamma|]\!] \to \mathbf{2}$, but it is more convenient to think of it as a subset of $[\![|\Gamma|]\!]$ – the set of contexts for which the proposition holds. So $\top$ is the whole set of contexts, $\bot$ is the empty set, and disjunction and conjunction are modelled by union and intersection. Because conjunction is written $u : \varphi \wedge \psi[u]$, we have to extend the environment of the interpretation of ψ. Since we erase all propositions to $\mathbf{1}$, we just choose ret $()$ as the erased proof. The same idea is used in the case of propositional implication. Quantifiers in our language of propositions are interpreted by quantifiers in the meta-language, and equality is interpreted as the set of contexts $G \in [\![|\Gamma^{\uparrow}|]\!]$ for which the equality holds. Note in particular that this means that the law of the excluded middle is sound, and cannot interfere with program execution in any way (since all propositions are erased).

Because the unrefined language is ambiently effectful (with the effect of error stops), every expression lies in a monadic type in the semantics. The refined semantics of the monad is given by the $\mathcal{E}[\![\Gamma \vdash A \text{ ty}]\!]$ relation, which picks out the subset of the monad not equal to error_A. That is, we require all refined terms to terminate without error.

Propositions are interpreted as a map from the intepretation of contexts into the (boolean, classical) truth values, or equivalently, as a subset of the contexts. Hence each logical connective is interpreted by the corresponding operation in the Boolean algebra of sets – conjunction is intersection, disjunction is union, and so on. We interpret equality at a type A as equality of the underlying elements of $|A|$.

This interpretation, while simple, is in some sense *too* simple – it causes function extensionality to fail. In particular, suppose $\text{Pos} \equiv \{x : \mathbb{Z} \mid x > 0\}$ is the type of positive numbers. Two functions

$f, g : \mathrm{Pos} \to \mathbb{Z}$ may agree on all positive arguments but fail to be equal because they erase to two functions which do different things on negative arguments. (For example, consider $f = \mathrm{id}$ and $g = \mathrm{abs}$.) This is also the reason that the η_{ir} and η_{pr} rules have an inhabitation premise (e.g., $\Gamma^{\uparrow} \vdash i : A$). A more complex semantics based on partial equivalence relations could let us resolve this issue, but we preferred to stick with the simplest possible semantics for expository reasons.

The semantics of types and propositions is defined over a bigger set of contexts than just the valid ones, but once we have this semantics, we can use it to define the valid contexts. Again, the interpretation $[\![\Gamma \text{ ok}]\!]$ is going to be the subset of $[\![|\Gamma^{\uparrow}|]\!]$ which satisfy all the propositions and in which all the values lie within their λ_{ert} types. So the empty context is inhabited by the empty environment; the context $\Gamma, u : \varphi$ is inhabited by $(G, \mathrm{ret}\ ())$ when G is in $[\![\Gamma \text{ ok}]\!]$ and φ is satisfied; and $\Gamma, x : A$ is inhabited by $(G, \mathrm{ret}\ x)$ when $G \in [\![\Gamma \text{ ok}]\!]$ and x is in the (subset) interpretation of A. (The case of ghost variables is the same as ordinary variables, since we care about the values of ghost variables when interpreting propositions in refined types.) It is easy to show that no λ_{ert} type can contain any errors.

Lemma 5.6 (Convergence). *Given an λ_{ert} derivation $\Gamma \vdash A \text{ ty}$, we have $\forall G, \mathrm{error} \notin [\![\Gamma \vdash A \text{ ty}]\!]\ G$*
Formalized as: `LogicalRefinement/Denot/Basic.lean`, `theorem HasType.denote_ty_non_null`

However, we give semantics to λ_{ert} *terms* by erasure, and so we have to connect the erased semantics of λ_{ert} terms to these semantic types. Furthermore, the semantics of λ_{ert} types cares about ghost values, but the semantics of erased terms ignore ghost values.

To relate these two, we also need to define the corresponding notion of a downgrade of an environment. Given an environment $G \in [\![|\Gamma^{\uparrow}|]\!]$, we recursively define its downgrade $G^{\downarrow}_{\Gamma} \in [\![|\Gamma|]\!]$, written $G^{\downarrow}$ when Γ is clear from context, as

$$*^{\downarrow}_{\cdot} = *, \qquad (x, G)^{\downarrow}_{u:\varphi,\Gamma} = (x, G^{\downarrow}_{\Gamma}), \qquad (y, G)^{\downarrow}_{x:A,\Gamma} = (y, G^{\downarrow}_{\Gamma}), \qquad (y, G)^{\downarrow}_{\|x:A\|,\Gamma} = (*, G^{\downarrow}_{\Gamma}) \tag{25}$$

Semantically, this discards all of the ghost information from an environment. We may now state the primary theorems proven about the semantics of λ_{ert}, namely, semantic substitution and semantic regularity.

Theorem 5.7 (Semantic Substitution). *Given λ_{ert} derivation $\Gamma \vdash A \text{ ty}$, λ_{ert} substitution $\Gamma \vdash \sigma : \Delta$, and valid environment $D \in [\![\Delta \text{ ok}]\!]$, we have*

$$[\![\Delta \vdash [\sigma]A \text{ ty}]\!]\ D = [\![\Gamma \vdash A \text{ ty}]\!]([\![|\Gamma^{\uparrow} \vdash \sigma^{\uparrow} : \Delta^{\uparrow}|]\!]\ D)$$

Formalized as: `LogicalRefinement/Denot/Subst.lean`, `theorem SubstCtx.subst_denot`

We can also show that for any well-typed λ_{ert} term, its erasure lies in the interpretation of the λ_{ert} type. This shows that every well-typed term satisfies the properties of its fancy type.

Theorem 5.8 (Semantic Regularity). *Given an λ_{ert} derivation $\Gamma \vdash a : A$, we have*

$$\forall G \in [\![\Gamma \text{ ok}]\!], [\![|\Gamma \vdash a : A|]\!]\ G^{\downarrow} \in [\![\Gamma \vdash A \text{ ty}]\!]\ G$$

Formalized as: `LogicalRefinement/Denot/Regular.lean`, `theorem HasType.denote`

Furthermore, it is an immediate corollary of lemma 5.6 and theorem 5.8 that the λ_{ert} logic is consistent, since $\{x : 1 \mid \phi\}$ is inhabited if and only if ϕ is true. It is also a corollary that for any well-typed term $\Gamma \vdash a : A$, we have that $\forall G \in [\![\Gamma \text{ ok}]\!], [\![|\Gamma \vdash a : A|]\!]\ G \neq \mathrm{error}$. That is, "well-typed programs do not go wrong".

$$\boxed{[\![\Gamma \vdash A \text{ ty}]\!] : [\![|\Gamma^{\uparrow}|]\!] \to \mathcal{P}([\![|A|]\!])}$$

$$\begin{aligned}
[\![\Gamma \vdash \mathbf{1} \text{ ty}]\!]\ G &= \{()\} \\
[\![\Gamma \vdash (x : A) \to B \text{ ty}]\!]\ G &= \{f \in [\![|A|]\!] \to \mathsf{M}[\![|B|]\!] \mid \forall x \in [\![\Gamma \vdash A \text{ ty}]\!]\ G, \\
&\qquad f\ x \in \mathcal{E}[\![\Gamma, x : A \vdash B \text{ ty}]\!]\ (G, \mathsf{ret}\ x)\} \\
[\![\Gamma \vdash (x : A) \times B \text{ ty}]\!]\ G &= \{(l, r) \mid l \in [\![\Gamma \vdash A \text{ ty}]\!]\ G \wedge r \in [\![\Gamma, x : A \vdash B \text{ ty}]\!]\ (G, \mathsf{ret}\ l)\} \\
[\![\Gamma \vdash A + B \text{ ty}]\!]\ G &= \mathsf{inl}([\![\Gamma \vdash A \text{ ty}]\!]\ G) \cup \mathsf{inr}([\![\Gamma \vdash B \text{ ty}]\!]\ G) \\
[\![\Gamma \vdash (u : \varphi) \Rightarrow A \text{ ty}]\!]\ G &= \{f \in \mathbf{1} \to \mathsf{M}[\![|A|]\!] \mid \\
&\qquad G \in [\![\Gamma \vdash \varphi \text{ pr}]\!] \implies f\ () \in \mathcal{E}[\![\Gamma, u : \varphi \vdash A \text{ ty}]\!]\ (G, \mathsf{ret}\ ())\} \\
[\![\Gamma \vdash \{x : A \mid \varphi\} \text{ ty}]\!]\ G &= \{a \in [\![\Gamma \vdash A \text{ ty}]\!]\ G \mid (G, a) \in [\![\Gamma, x : A \vdash \varphi \text{ pr}]\!]\} \\
[\![\Gamma \vdash \forall x : A, B \text{ ty}]\!]\ G &= \{f \in \mathbf{1} \to \mathsf{M}[\![|B|]\!] \mid \\
&\qquad \forall x \in [\![\Gamma \vdash A \text{ ty}]\!]\ G, f\ () \in \mathcal{E}[\![\Gamma, x : A \vdash B \text{ ty}]\!]\ (G, \mathsf{ret}\ x)\} \\
[\![\Gamma \vdash \exists x : A, B \text{ ty}]\!]\ G &= \bigcup_{x \in [\![\Gamma \vdash A \text{ ty}]\!]\ G} [\![\Gamma, x : A \vdash B \text{ ty}]\!](G, \mathsf{ret}\ x) \\
[\![\Gamma \vdash \mathbb{N} \text{ ty}]\!]\ G &= \mathbb{N} \\
\mathcal{E}[\![\Gamma \vdash A \text{ ty}]\!] &= \lambda G, \mathsf{ret}\ ([\![\Gamma \vdash_{\lambda} A \text{ ty}]\!]\ G) : [\![|\Gamma^{\uparrow}|]\!] \to \mathcal{P}(\mathsf{M}[\![|A|]\!])
\end{aligned}$$

$$\boxed{[\![\Gamma \vdash \varphi \text{ pr}]\!] : \mathcal{P}([\![|\Gamma^{\uparrow}|]\!])}$$

$$\begin{aligned}
[\![\Gamma \vdash \top \text{ pr}]\!] &= [\![|\Gamma^{\uparrow}|]\!] \\
[\![\Gamma \vdash \bot \text{ pr}]\!] &= \varnothing \\
[\![\Gamma \vdash (u : \varphi) \Rightarrow \psi \text{ pr}]\!] &= \{G \mid G \in [\![\Gamma \vdash \varphi \text{ pr}]\!] \implies (G, \mathsf{ret}\ ()) \in [\![\Gamma, u : \varphi \vdash \psi \text{ pr}]\!]\} \\
[\![\Gamma \vdash (u : \varphi) \wedge \psi \text{ pr}]\!] &= \{G \mid G \in [\![\Gamma \vdash \varphi \text{ pr}]\!] \wedge (G, \mathsf{ret}\ ()) \in [\![\Gamma, u : \varphi \vdash \psi \text{ pr}]\!]\} \\
[\![\Gamma \vdash \varphi \vee \psi \text{ pr}]\!] &= [\![\Gamma \vdash \varphi \text{ pr}]\!] \cup [\![\Gamma \vdash \psi \text{ pr}]\!] \\
[\![\Gamma \vdash \forall x : A, \varphi \text{ pr}]\!] &= \{G \mid \forall x \in [\![\Gamma \vdash A \text{ ty}]\!]\ G, (G, \mathsf{ret}\ x) \in [\![\Gamma, x : A \vdash \varphi \text{ pr}]\!]\} \\
[\![\Gamma \vdash \exists x : A, \varphi \text{ pr}]\!] &= \{G \mid \exists x \in [\![\Gamma \vdash A \text{ ty}]\!]\ G, (G, \mathsf{ret}\ x) \in [\![\Gamma, x : A \vdash \varphi \text{ pr}]\!]\} \\
[\![\Gamma \vdash a =_A b \text{ pr}]\!] &= \{G \mid [\![|\Gamma^{\uparrow} \vdash a : A|]\!]\ G = [\![|\Gamma^{\uparrow} \vdash b : A|]\!]\ G\}
\end{aligned}$$

$$\boxed{[\![\Gamma \text{ ok}]\!] : \mathcal{P}([\![|\Gamma^{\uparrow}|]\!])}$$

$$\begin{aligned}
[\![\cdot \text{ ok}]\!] &= \{\cdot\} \\
[\![\Gamma, x : A \text{ ok}]\!] = [\![\Gamma, \|x : A\| \text{ ok}]\!] &= \{(G, \mathsf{ret}\ x) \mid G \in [\![\Gamma \text{ ok}]\!] \wedge x \in [\![\Gamma \vdash A \text{ ty}]\!]\ G\} \\
[\![\Gamma, u : \varphi \text{ ok}]\!] &= ([\![\Gamma \text{ ok}]\!] \cap [\![\Gamma \vdash \varphi \text{ pr}]\!]) \times \mathsf{M}\mathbf{1}
\end{aligned}$$

Fig. 14. Denotations for λ_{ert}

6 FORMAL VERIFICATION

We have proved all the results stated in the previous sections in Lean 4. The proof development is about 15.8 kLoC in length and is partially automated, though there is much potential for further automation. In particular, lemmas 4.1, 4.2, and 4.3 are heavily automated, with one template tactic that applies to most of the cases, whereas theorems 5.7 and 5.8 and lemmas 5.2, 5.4, and 5.5 had to be proved manually. These properties typically had lots of equality coercions, which had to be plumbed manually because they are difficult to automate. It is possible that with more experience with Lean tactics they might also be automated, but we were not able to do so.

The formalized syntax and semantics are mostly the same as that presented in this writeup. There are two main differences. First, we have implemented variables using de-Bruijn indices. Second, we folded types, propositions, terms, and proofs into a single inductive type to avoid mutual recursion, which Lean 4 currently has poor support for.

This project was the first time the authors used Lean 4 for serious formalization work. While we ran into numerous issues due to Lean still being in active early-stage development, we found it to be a highly effective formalization tool. One issue we ran into was very high memory usage and, in some cases, timeouts, when using Lean's `simp` tactic on complex pattern matches. The addition of the `dsimp` tactic, after a discussion on the Lean 4 Zulip, mostly alleviated this, and performance has improved in later versions of Lean. We otherwise found the quality of automation to be very good: even though the authors are novices at Lean, we were able to easily maintain and extend the proofs without needing to edit theorems proved via tactics. For example, we originally forgot to include the Unit-WF axiom, but we were able to include it with only minor edits to the manual theorems in about 30 minutes. As the formalization made heavy use of dependent types, we also ran into many issues attempting to establish equalities between dependently typed terms. However, in this case, we found Lean relatively easy to use compared to other dependently-typed proof assistants based on dependent types, with our experiments at Coq-based formalization running into similar issues.

7 DISCUSSION

Function Extensionality, Recursion and Effects. Our current semantics is inconsistent with function extensionality because two functions must be equal over their entire, unrefined domain to satisfy the denotation of the equality type. To support extensionality (and related types like quotient types), we should be able to intepret the calculus via a semantics based on partial equivalence relations, as in [Harper 1992].

We also want to reason about the partial correctness and divergent programs. Hence, it makes sense to add support for general recursive definitions, including nonterminating definitions, by moving to a domain-theoretic semantics (rather than the set-theoretic semantics we currently use). We also want to extend the base language with more effects (such as store and IO) and extend λ_{ert} to support fine-grained reasoning about them via an effect system such as in [Katsumata 2014].

Categorical Semantics. The motivating model of refinement types underlying our work is that of [Melliès and Zeilberger 2015], which equates type refinement systems with functors from a category of typing derivations to a category of terms. In essence, one can view our work as taking the setup in [Melliès and Zeilberger 2015] and inlining all the categorical definitions for the case of the simply-typed lambda calculus.

We would like to update our formalisation to work in terms of the categorical semantics; this would let us to account for all of the extensions above at once, without having to reprove theorems (such as semantics substitution and regularity) for each modification.

Recently, Kura [2021] has studied the denotational semantics of a system of refinements over Ahman's variant of dependent call-by-push-value [Ahman 2018]. Kura also uses a fibrational semantics similar to the Zeilberger-Mellies semantics (as well as Katsumata's semantics of effect systems [Katsumata 2014]) to equip a dependent type theory with a subtype relation arising from the entailment relation of first-order logic. Unlike λ_{ert}, the use of subtyping means that there are no explicit proofs, and hence type checking is undecidable.

Units in the Erasure. The erasure function for λ_{ert}introduces many units as part of the translation (e.g., $|u : \phi \Rightarrow A| = \mathbf{1} \rightarrow |A|$). This is necessary to ensure the property that no well-typed term can signal an error. Consider the example:

$$\hat{\lambda}||u : \bot||.\text{absurd}(u) : (u : \bot) \Rightarrow A$$

This erases to the term:

$$\lambda x : \mathbf{1}.\text{error}$$

Without the $\lambda x : \mathbf{1}$, the term would erase to error, which violates our invariant.

It is possible to avoid introducing any units at all if the base calculus is in call-by-push-value form. However, since one of our main objectives was to have a simple, familiar semantics, we the price of introducing units was lower than introducing call-by-push-value.

Automation and Solver Integration. One of the critical advantages of refinement types is the potential for significantly reducing the annotation burden of formal verification. Hence, to make λ_{ert} usable, it should be able to be automated to a similar degree for similarly complex programs. One potential form of basic automation is support for an "smt" tactic, similar to section 3's "β" tactic; we can similarly envision calling out to various automated theorem provers like Vampire [Kovács and Voronkov 2013] or SPASS [Weidenbach et al. 2002].

A more powerful approach would be to adapt the work on Liquid Typing [Rondon et al. 2008] to this setting, which works by inferring appropriate refinement types and proofs for an unrefined program such that, given that the program's preconditions are satisfied, the preconditions of all function calls within the program as well as the postconditions of the program are both satisfied. Liquid Typing sometimes requires annotations to infer appropriate invariants and may require explicit checks to be added for conditions it cannot verify are implied by the preconditions. One way to combine Liquid Typing with λ_{ert} would be to, using λ_{ert} types as annotations, automatically refine the types of subterms of an λ_{ert} program to make it typecheck, inferring and inserting proofs as necessary. One advantage of this approach would be that (assuming it compiles down to fully-annotated λ_{ert}) it removes the liquid typing algorithm itself from the trusted codebase, and, if the solvers used support proof output, the solvers themselves as well. Furthermore, we could replace potentially expensive runtime checks with explicit proofs.

8 RELATED WORK

8.1 Relationship of λ_{ert} to Dependent Types

The most well-known approach to designing programming languages with integrated support for proof is dependent type theory. The semantics of dependent type theories is generally given in an "intrinsic" style, in which well-typed terms (in fact, typing derivations) are given a denotational semantics, and ill-typed terms are regarded as meaningless (i.e., have no semantics).

On the other hand, λ_{ert} is a refinement type system, which takes an existing programming language (in this case, the simply-typed lambda calculus with error stops), and extends it so that existing programs can be given richer, more precise types. This ensures that it is always possible to forget the rich types and be left with the simply-typed skeleton. A good analogy is to Hoare logic, in which pre-conditions, post-conditions and loop invariants can be seen as rich type annotations on a simple while-program. These logical annotations can always be erased, leaving behind an untyped while-program.

Like in Hoare logic, λ_{ert} distinguishes logical assertions from the type-theoretic structure of the programming language. This is in contrast to the traditional Curry-Howard interpretation of logic in dependent type theory, and more closely resembles the Prop type of Coq (which is a sort of purely logical assertions), or even more closely the "logic-enriched type theories" of Aczel and

Gambino [Gambino and Aczel 2006] (which extends type theory with a new judgement of logical propositions).

Most dependent type theories also feature a notion of judgemental equality, in which types are considered equal modulo some equational theory (usually containing β and sometimes η-conversions). λ_{ert} is designed to work without a judgemental equality, since it is a source of both metatheoretical difficulty, and complicates the design of tooling. However, there are some dependent type theories, such as Objective Type Theory [van den Berg and den Besten 2021] and Zombie [Sjöberg and Weirich 2015], which implement reduction propositionally as axioms, similarly to what we have done.

8.2 Refinement Logics and Squashed Curry-Howard

As is well-known, even type theories without a Prop sort like Coq or Lean's still have a logical reading – the famous "propositions as types" principle, where each type-theoretic connective (function types, pairs, sums) corresponds to a logical connective (implication, conjunction, disjunction).

This is unsuitable for our purposes. We plan to use λ_{ert} as the basis of extending practical SMT-based refinement type systems with explicit proofs, and SMT solvers are fundamentally based on classical logic. So we want the semantics of the propositions in our refinement types to be classical as well, which is not possible when propositions and types are identified.

This also makes it difficult to use the various modal techniques proposed to integrate proof irrelevance into type theory, such as Awodey and Bauer's squash types [Awodey and Bauer 2004], or Sterling and Harper's phase modalities [Sterling and Harper 2021].

For example, with the Awodey-Bauer squash type $[A]$, logical disjunctions and existentials are encoded as

$$\begin{aligned} P \vee Q &= [P + Q] \\ \exists x : X.P &= [\Sigma x : X.P] \end{aligned} \tag{26}$$

where the $[-]$ operator has a degenerate equality. As a result, if the type theory is intuitionistic the logic of propositions must be as well, which is contrary to our needs.

However, there is a deeper problem with using squash types. The semantics of the Awodey-Bauer squash type is such that if P is a proposition (i.e., all inhabitants are equal), then P and $[P]$ are isomorphic. Since P is a proposition if all its inhabitants are equal, a contractible type like $\Sigma x : \mathbb{N}.\, x = n$ is a proposition, and hence there is a map $[\Sigma x : \mathbb{N}.x = n] \longrightarrow \Sigma x : \mathbb{N}.x = n$.

That is, it is possible to extract computational data from a squashed type, and so erasure of propositions and squashed types is a much more subtle problem than it may first seem. Kraus *et al*'s paper *Notions of Anonymous Existence in Martin-Löf Type Theory* [Kraus et al. 2017] studies this and similar issues in detail. There is a similar obstacle when using the Sterling-Harper approach, which could use a pair of modalities to control whether a term is potentially in the specification or runtime phases. Once again, re-using type-theoretic connectives as logical connectives forces the identification of the refinement logic and the type theory.

8.3 Erasure in Dependent Type Theory

Coq [The Coq Development Team 2021] supports a notion of erasure, in which terms of proposition type are systematically elided as a dependently-typed program is extracted to a functional language like Ocaml or Haskell. It also lets users declare certain function parameters as useless, but since Coq does not distinguish logical variables from computational ones in its type system, a well-typed Coq term may fail to successfully extract. In contrast, λ_{ert} can always erase both proofs and logical variables, and furthermore guarantees that all erased terms are also well-typed because it is a refinement over a pre-existing simple type theory. This is not the case in Coq, where extracted terms may have to use the unsafe cast Obj.magic.

Another critical feature of λ_{ert} is that it supports Hoare-style logical variables. is that they are not program variables, and cannot influence the runtime behaviour of a program – they only exist for specification purposes. So in a type

$$\mathsf{vlen} : \forall n : \mathbb{N}. \mathsf{Vec}\ n \longrightarrow \{x : \mathbb{N} \mid x = n\} \tag{27}$$

where $\forall n : \mathbb{N}$ is an intersection-style quantifier, it is only possible to compute the length of the list by actually traversing the list. In plain Martin Löf type theory, the corresponding type has a degenerate implementation:

$$\begin{aligned} &\mathsf{vlen} : \Pi n : \mathbb{N}. \mathsf{Vec}\ n \longrightarrow \Sigma x : \mathbb{N}. x = n \\ &\mathsf{vlen}\ n\ \_ \equiv (n, \mathsf{refl}) \end{aligned} \tag{28}$$

This kind of computational irrelevance is different from what is sometimes called proof-irrelevance or definitional irrelevance, since different values of n are not equal.

Two approaches that have arisen to manage this kind of irrelevance are the implicit forall quantifier of the implicit calculus of constructions (ICC) [Barras and Bernardo 2008], and the usage annotations in Atkey and McBride's quantitative type theory (QTT) [Atkey 2018].

As we did, the ICC introduced an intersection type $\forall x : T.U$ to support computationally irrelevant quantification. However, because there was no separation of the refinement layer from the base layer, the denotational semantics of the ICC is much more complicated – the Luo-style extended calculus of constructions (ECC) [Luo 1990] has a simple set-theoretic model, but the only known denotational model of the ICC [Miquel 2000] is based on coherence spaces. In our view, this is a significant increase in the complexity of the model.

In contrast, we are able to model intersection types with ordinary set-theoretic intersections. The reason for this is that the refinement type discipline ensures that we only ever make use of *structural* set-theoretic operations. That is, every λ_{ert} type is a subset of an underlying base type, and so when we take an intersection, we are only taking the intersection of a family of subsets of a particular set (the base type). From a mathematical point of view, this is much better-behaved than taking intersections of *arbitrary* sets, and having this invariant lets us interpret intersections more simply than is possible in the semantics of the ICC.

Our semantics (and indeed, syntax) for intersection types is very similar to the semantics of the "dependent intersection types" introduced by Kopylov [2003] for the Nuprl system. They worked with partial equivalence relations over a term model, rather than our simple set-theoretic model. As mentioned above, we will also need to move to a PER model to support function extensionality, though we expect we can still consider PERs over sets, rather than having to use a term model.

In his PhD dissertation [Tejiščák 2019], Tejiščák studies how to apply QTT directly to the problem of managing computational irrelevance. The runtime and erasable annotations in QTT look very similar to our distinction between computational and ghost variables. However, we cannot directly use this type system, because it does not have a sort of propositions, and therefore must express logical properties in a construtive, Curry-Howard logic.

8.4 Relation to Other Systems

ATS [Cui et al. 2005] is a system in the Dependent ML style. That is, the type system has a very hard separation between indices (which can occur inside types) and terms (which do runtime computation). Like λ_{ert}, it is possible in ATS to give explicit proofs of quantified and inductive formulae.

ATS's erasure theorem is proved operationally, by exhibiting a simulation between the reductions of the fully-typed language and erased, untyped programs. This makes it hard to reason about the equality of program terms (something like a logical relation would have to be added), but ATS does not have to, because it distinguishes program terms and indices, and only permits indices to occur in types.

However, since program terms cannot occur in constraints, correctness arguments about functions are forced into an indirect style – for each recursive function, one must define an inductive relation encoding its graph, and then show that inputs and outputs are related according to this relation. As a result, proving something like (e.g.) that the type of endofunctions $A \to A$, the identity, and function composition have the structure of a monoid will be very challenging. In λ_{ert}, in contrast, this would be very straightforward, since (following liquid types) indices and program terms are one and the same.

Modern ATS has also extended the proposition language to support a notion of linear assertion, which permits verifying imperative programs in the style of separation logic. Extending λ_{ert} with support for richer reasoning about effectful programs is ongoing work.

F*is a full dependent type theory which has replaced the usual conversion relation with an SMT-based approach. F*makes no effort to keep quantifiers out of the constraints sent to its SMT solver, and hence does not (and cannot) have any decidability guarantees – the F*typechecking problem is undecidable. This best-effort view lets F*take maximum advantage of the solver, at the price of sometimes letting the typechecker loop. In contrast, λ_{ert} has decidable, near linear-time typechecking, because typing is fully syntax-directed and has no conversion relation.

The treatment of computational irrelevance in F*is similar in effect (though different in technical detail) to ICC. As in ICC, ghost arguments can affect typing but not computations, but there is no notion of a less-typed base language that ghosts can be erased to.

REFERENCES

Danel Ahman. 2018. Handling fibred algebraic effects. *Proc. ACM Program. Lang.* 2, POPL (2018), 7:1–7:29. https://doi.org/10.1145/3158095

Robert Atkey. 2018. The Syntax and Semantics of Quantitative Type Theory. In *LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom.* https://doi.org/10.1145/3209108.3209189

Steven Awodey and Andrej Bauer. 2004. Propositions as [types]. *Journal of logic and computation* 14, 4 (2004), 447–471.

Bruno Barras and Bruno Bernardo. 2008. The implicit calculus of constructions as a programming language with dependent types. In *Foundations of Software Science and Computational Structures: 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 11.* Springer, 365–379.

Sa Cui, Kevin Donnelly, and Hongwei Xi. 2005. Ats: A language that combines programming with theorem proving. In *Frontiers of Combining Systems: 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005. Proceedings 5.* Springer, 310–320.

Nicola Gambino and Peter Aczel. 2006. The generalised type-theoretic interpretation of constructive set theory. *The Journal of Symbolic Logic* 71, 1 (2006), 67–103. https://doi.org/10.2178/jsl/1140641163

Robert Harper. 1992. Constructing type systems over an operational semantics. *Journal of Symbolic Computation* 14, 1 (1992), 71–84. https://doi.org/10.1016/0747-7171(92)90026-Z

Ranjit Jhala and Niki Vazou. 2020. Refinement Types: A Tutorial. *arXiv e-prints*, Article arXiv:2010.07763 (Oct. 2020), arXiv:2010.07763 pages. arXiv:2010.07763 [cs.PL]

Shin-ya Katsumata. 2014. Parametric Effect Monads and Semantics of Effect Systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 633–645. https://doi.org/10.1145/2535838.2535846

Alexei Kopylov. 2003. Dependent Intersection: A New Way of Defining Records in Type Theory. In *18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada, Proceedings.* IEEE Computer Society, 86–95. https://doi.org/10.1109/LICS.2003.1210048

Laura Kovács and Andrei Voronkov. 2013. First-Order Theorem Proving and Vampire. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–35.

Nicolai Kraus, Martín Escardó, Thierry Coquand, and Thorsten Altenkirch. 2017. Notions of anonymous existence in Martin-Löf type theory. *Logical Methods in Computer Science* 13, 1 (2017).

Satoshi Kura. 2021. A General Semantic Construction of Dependent Refinement Type Systems, Categorically. In *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European*

Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12650), Stefan Kiefer and Christine Tasson (Eds.). Springer, 406–426. https://doi.org/10.1007/978-3-030-71995-1_21

Zhaohui Luo. 1990. *An extended calculus of constructions*. Ph. D. Dissertation. University of Edinburgh.

Paul-André Melliès and Noam Zeilberger. 2015. Functors Are Type Refinement Systems. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 3–16. https://doi.org/10.1145/2676726.2676970

Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* 17, 3 (1978), 348–375.

Alexandre Miquel. 2000. A model for impredicative type systems, universes, intersection types and subtyping. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No. 99CB36332)*. IEEE, 18–29.

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4 Selections from 1989 IEEE Symposium on Logic in Computer Science.

John C. Reynolds. 2003. *What do types mean? — From intrinsic to extrinsic semantics*. Springer New York, New York, NY, 309–327. https://doi.org/10.1007/978-0-387-21798-7_15

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. *SIGPLAN Not.* 43, 6 (jun 2008), 159–169. https://doi.org/10.1145/1379022.1375602

Vilhelm Sjöberg and Stephanie Weirich. 2015. Programming up to Congruence. *SIGPLAN Not.* 50, 1 (Jan. 2015), 369–382. https://doi.org/10.1145/2775051.2676974

Jonathan Sterling and Robert Harper. 2021. Logical relations as types: Proof-relevant parametricity for program modules. *Journal of the ACM (JACM)* 68, 6 (2021), 1–47.

Matúš Tejiščák. 2019. *Erasure in Dependently Typed Programming*. Ph. D. Dissertation. University of St. Andrews.

The Coq Development Team. 2021. *The Coq Proof Assistant*. LogiCal Project. https://doi.org/10.5281/zenodo.4501022

Benno van den Berg and Martijn den Besten. 2021. Quadratic type checking for objective type theory. *arXiv e-prints*, Article arXiv:2102.00905 (Feb. 2021), arXiv:2102.00905 pages. arXiv:2102.00905 [cs.LO]

Niki Vazou and Michael Greenberg. 2022. How to safely use extensionality in Liquid Haskell. In *Haskell '22: 15th ACM SIGPLAN International Haskell Symposium, Ljubljana, Slovenia, September 15 - 16, 2022*, Nadia Polikarpova (Ed.). ACM, 13–26. https://doi.org/10.1145/3546189.3549919

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. *SIGPLAN Not.* 49, 9 (aug 2014), 269–282. https://doi.org/10.1145/2692915.2628161

Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobald, and Dalibor Topić. 2002. Spass Version 2.0. In *Automated Deduction—CADE-18*, Andrei Voronkov (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 275–279.

Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. 4, OOPSLA, Article 193 (nov 2020), 25 pages. https://doi.org/10.1145/3428261

Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking through Dependent Types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 249–257. https://doi.org/10.1145/277650.277732

Received 2023-03-01; accepted 2023-06-27