PLEX: Normalization for Refinement Types

ALESSIO FERRARINI, IMDEA Software Institute, Spain and Universidad Politécnica de Madrid, Spain
NIKI VAZOU, IMDEA Software Institute, Spain
WOUTER SWIERSTRA, Utrecht University, Netherlands

Refinement types often use SMT solvers to automate program verification. However, since SMT solvers are first-order, verification of properties that requires higher-order reasoning is not possible. Proof by Logical Evaluation (PLE) is an algorithm that provides a layer between refinement types and SMT solvers that permits symbolic evaluation of functions, but it lacks support for higher-order reasoning. We introduce PLEX, an extension to PLE, that supports η-expansions, β-reductions, and dependent pattern matching. We prove that PLEX is sound and terminating, describe its implementation in Liquid Haskell, and evaluate it on examples that make essential use of higher-order data, and as such they cannot be handled by PLE. The new PLEX algorithm bridges the gap between higher-order languages and first-order SMT solvers via refinement types.

CCS Concepts: • **Theory of computation** → **Program verification**; • **Software and its engineering** → *General programming languages.*

Additional Key Words and Phrases: refinement types, SMT-based verification, higher-order reasoning

ACM Reference Format:
Alessio Ferrarini, Niki Vazou, and Wouter Swierstra. 2026. PLEX: Normalization for Refinement Types. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 140 (April 2026), 27 pages. https://doi.org/10.1145/3798248

1 Introduction

Refinement types [Jhala and Vazou 2021] constrain the types of a programming language with logical predicates to express verification properties. For example, the refined type `{v:Int | 0 /= v}` describes integer values that are different from zero, thus can be used to prevent division by zero. Various programming languages, *e.g.*, Haskell [Vazou et al. 2014], Java [Gamboa et al. 2023], and Rust [Lehmann et al. 2023], have been extended with refinement types to verify a variety of properties, ranging from safe-indexing [Xi and Pfenning 1998] to security [Bengtson et al. 2011].

Refinement types are successful, partly because verification is highly automated by SMT solvers. Essentially, a refinement type system reduces a verification problem into a set of logical formulas, whose validity is then checked by an SMT solver. These SMT solvers are very efficient in deciding satisfiability (thus also validity) of formulas over first-order theories, such as linear arithmetic, arrays, and uninterpreted functions. However, they are not designed to reason about higher-order functions, which appear ubiquitously in functional programming languages.

Several approaches aim to bridge this gap. For example, `Dafny` [Leino 2010] and `F*` [Swamy et al. 2016] encode functions into the SMT logic using universally quantified axioms of the form $\forall x_1\ x_2\ x_3 . f\ x_1\ x_2\ x_3\ = e$, where any higher-order function has been defunctionalized [Reynolds 1972]. These axioms are then fed to the SMT solver, leading to automatic and precise verification; but potentially undecidable SMT queries [Leino and Pit-Claudel 2016]. Refinement reflection [Vazou

Authors' Contact Information: Alessio Ferrarini, IMDEA Software Institute, Madrid, Spain and Universidad Politécnica de Madrid, Madrid, Spain, alessio.ferrarini@imdea.org; Niki Vazou, IMDEA Software Institute, Madrid, Spain, niki.vazou@imdea.org; Wouter Swierstra, Utrecht University, Utrecht, Netherlands, w.s.swierstra@uu.nl.

ACM 2475-1421/2026/4-ART140
https://doi.org/10.1145/3798248

et al. 2017], on the other hand, provides an alternative approach. Instead of feeding universally quantified axioms to the SMT solver, it preprocesses these axioms using Proof by Logical Evaluation (PLE), a symbolic evaluation algorithm that instantiates the axioms based on the program's structure. The PLE algorithm itself uses the SMT solver and the evaluation context to decide if a function, encoded as an axiom, can be instantiated without leading to infinite loops.

Even though PLE is effective in practice and provably terminating, it cannot always reason about higher-order programs. The axioms are only triggered when functions are fully applied. For example, since f takes three arguments, PLE can expand $f\ x_1\ x_2\ x_3$ to e, but will not expand $f\ x_1\ x_2$ to $\lambda y.e\ y$. Of course, such expansions are not trivial: they require reasoning about higher-order lambda terms in the first-order setting of SMT solvers. Furthermore, PLE does not support unification, which is often essential to instantiate the axioms. For example, suppose that the SMT solver can prove that $f = g$, assuming a defunctionalized encoding of the functions f and g. Without unification, this equality alone is not sufficient to trigger the unfolding of $g\ 1\ 2\ 3$ to $e[1/x_1][2/x_2][3/x_3]$.

We present PLEX, a novel algorithm that extends PLE to support higher-order reasoning, η and β rules, together with equalities that arise from unification during dependent pattern matching. We implemented the extension on top of PLE and used it to successfully verify `Liquid Haskell` programs. In particular, PLEX has been used to verify higher-order programs that required η equivalence and unification, without the need for manual proofs or adding axioms (potentially inconsistent with refinement types) such as functional extensionality [Vazou and Greenberg 2022]. Thus, we conclude that PLEX is a significant step in the automation of higher-order program verification using SMT based methods. Concretely, we make the following contributions:

- First, we formalize the concept of dependent pattern matching in the context of refinement types (§ 3), show that it is a generalization of higher-order unification, and provide an approximation that is both sound and computable (Theorem 3.4).
- Second, we extend the PLE algorithm with the η and β equivalence rules, as well as a novel rule, that we call δ, that unfolds function definitions using the unifiers provided by the pattern matching (§ 4). We prove that the extension is sound and terminating (Theorems 4.3 and 4.4). Crucially, to establish soundness of the final algorithm, we define a delicate notion of function equality (§ 3.1.2), that takes into account the refined domains of the functions.
- We implemented PLEX as an extension of `Liquid Haskell`'s PLE algorithm and used it to verify eight examples that all make essential use of higher-order functions (§ 5). Compared to the original PLE algorithm, our development is *more precise*, since our higher-order examples were not possible to verify before. The price to pay for this added expressivity is modest: the PLEX algorithm is only 5% slower on the existing test suite of `Liquid Haskell`. Compared to `Agda`, our examples are very similar in size and structure; while each tool has its own strengths and weaknesses, *e.g.,* PLEX automates arithmetic reasoning and supports subtyping, facilitating certain proof steps that are typically more difficult in interactive proof assistants.

2 Overview

To illustrate the capabilities and design of PLEX, we implement a simple expression language and its compilation into stack machine operations, à la Bahr and Hutton [2015] and Pickard and Hutton [2021], and prove its correctness using `Liquid Haskell`. This example shows how refinement types (§ 2.1) can enforce light correctness guarantees and how data propositions (§ 2.2) can capture the semantics of programs. We then show how the PLEX algorithm can reason about higher-order functions (§ 2.3) while preserving the first-order automation of the SMT solver (§ 2.4).

2.1 Refinement Types: Calculating Correct Compilers

We start by defining the expression and the stack machine languages of our compilation.

The Expression Language. The expression language contains arithmetic expressions with integer constants, addition, multiplication, and negation, and is defined as the Haskell data type `Expr`:

```
data Expr = EConst Int | EAdd Expr Expr | EMul Expr Expr | ENeg Expr
```

The semantics of the expression language is defined by the `eval` function, which evaluates an expression to an integer:

```
eval :: Expr → Int  {-@ reflect eval @-}
eval (EConst x)   = x
eval (EAdd e1 e2) = eval e1 + eval e2
eval (EMul e1 e2) = eval e1 * eval e2
eval (ENeg e)     = - (eval e)
```

The `reflect eval` annotation is a `Liquid Haskell` annotation that instructs the system to reflect the definition of `eval` into the logic. Concretely, the type checking rules of `Liquid Haskell` (as defined in [Vazou et al. 2017] and extended in § 3) perform three actions when they encounter a `reflect` annotation: First, they generate an uninterpreted logical function *eval* that can be used in the refinements. Second, they check that the definition of `eval` is terminating. Finally, they pass the user definition of the `eval` function to the proof by evaluation (PLE) algorithm, to ensure that the logical function *eval* approximates the user definition. Later we explain how PLE works and how we extended it to more precisely approximate the user definitions.

The Stack Machine Language. The target language of our compilation is a stack machine that has three operations: to push a constant on the stack and to add or multiply the top two elements.

```
data OpCode = OpPush Int | OpAdd | OpMul
type Stack  = [Int]
```

The semantics of the stack machine is defined by the `execOpCode` function, that takes as argument an operation and transforms the stack accordingly.

```
execOpCode :: OpCode → Stack → Stack {-@ reflect execOpCode @-}
execOpCode (OpPush x) = push x -- push x xs = x:xs
execOpCode OpAdd      = \case (x : y : xs) → (y + x) : xs
execOpCode OpMul      = \case (x : y : xs) → (y * x) : xs
```

When the input operation is `OpPush x`, it simply calls the `push` function that puts the constant `x` on top of the stack. When the input operation is `OpAdd` or `OpMul`, it pops the top two elements of the stack, performs the respective operation, and pushes the result back on the stack. To ensure that the stack has enough elements to perform the operation, we define the `minStackSize` function that returns the minimum number of elements required by an operation.

```
minStackSize :: OpCode → Int  {-@ reflect minStackSize @-}
```

Having defined (and reflected into the logic) the `minStackSize` function, we can use it to refine the type of `execOpCode` to ensure that the function is well defined:

```
{-@ execOpCode :: o:OpCode → s:{ Stack | minStackSize o ≤ len s } → Stack @-}
```

This refinement type states that the length of the stack `s` must be at least `minStackSize o`, where `len` is a standard inductively defined length function that is also reflected into the logic by `Liquid Haskell`. This refinement type ensures that when the operation `o` is `OpAdd` or `OpMul`, the stack `s` has at least two elements, thus the definition of `execOpCode` is exhaustive.

2.2 Data Propositions: Programs for the Stack Machine

Next, we define the type of programs for the stack machine, which is essentially a list of stack operations. However, since we aim to build correct-by-construction compilers, we cannot use just any list. Instead, following similar approaches [Swierstra 2023] in dependently typed languages, our program type should be indexed by a function that captures the program's semantics.

In `Liquid Haskell`, indexed types can be encoded using *Data Propositions* [Borkowski et al. 2024]. Concretely, `Liquid Haskell`'s built-in `Prop` type constructor encodes propositions, since given any expression `e`, `Prop e` denotes that the proposition `e` holds. In `Liquid Haskell`, `Prop` is defined via the `measure` `prop`, that in the logic defines an uninterpreted function, as follows:

```
{-@ type Prop e = {v:a | e = prop v} @-}  {-@ measure prop :: a → b @-}
```

Using data propositions, we can define the type of programs for the stack machine as follows:

```
{-@ data Program where
      PNil  :: Prop (Program id)
      PCons :: op:OpCode → p:(Stack → { v:Stack | minStackSize op ≤ len v})
            → Prop (Program p) → Prop (Program (execOpCode op . p))            @-}
data PROGRAM = Program (Stack → Stack)
```

The program data type has two constructors:

- `PNil` constructs an empty program that does not perform any operations on the stack, so its index function is `id`. The type of `PNil` is `Prop (Program id)`, or equivalently, `{v:Program | prop v = Program id}`. *Note* that `Program` is overloaded to be both the defined type constructor and, inside the refinement, the data constructor indexed by a `Stack → Stack` function, corresponding to the semantics of the program.
- `PCons` appends a new operation to an existing program. It takes three arguments: an operation `op`; a function on stacks `p` that given any stack produces a stack that has at least `minStackSize op` elements; and a program that has semantics `p`. `PCons` constructs a composite program that has the semantics of running `p` followed by executing `op`. As a result, the return type of `PCons` is the composition of the operation and program `p`, that is `Prop (Program (execOpCode op . p))`.

This design serves two purposes. First, it ensures the construction of only well-defined programs. Before adding an operation, we confirm that the program produces enough elements on the stack. Second, it enables the design of a correct-by-construction compiler by explicitly constructing the semantics of a compiled program in its type. Our definition is very similar to the `Agda` version; we give a more extensive comparison between the two in § 5.4. The main difference is that in `Agda`'s version of the `PCons` constructor, the arguments `op` and `p` are left implicit.

2.2.1 Reasoning about Higher-Order Indices. Since stack machine programs are indexed by functions, verification of programs involves reasoning about higher-order functions. The vanilla PLE algorithm, up to now, did not support such kind of reasoning. To illustrate its limitations and motivate the need for our new algorithm, consider the `compose` function that composes two programs.

```
{-@ compose :: s1:(Stack → Stack) → s2:(Stack → Stack) → p1:Prop (Program s1)
            → p2:Prop (Program s2) → Prop (Program (s2 . s1)) @-} {-@ reflect compose @-}
compose :: (Stack → Stack) → (Stack → Stack) → Program → Program → Program
compose s1 s2 p1 p2 = case p2 of
  PNil                → p1
  PCons cmd srest rest → PCons cmd (srest . s1) (compose s1 srest p1 rest)
```

The `compose` function composes two programs `p1` and `p2`, whose semantics are captured by the functions `s1` and `s2`, *resp.*, and returns the composition of the two programs. The refinement type states that the resulting program has the semantics of `s2 . s1`. Using only PLE, `Liquid Haskell`

cannot verify the correctness of `compose`, as it involves reasoning about higher-order functions. In the rest of this section, we describe our modifications that render this verification possible.

Type checking starts by extending the typing environment with the types of the four arguments, as well as the type of the recursive function `compose` itself, thus generating the following environment. (We omit the bindings that are not relevant for the verification and on the right we expand the definition of the `Prop` type).

$$\Gamma \supseteq \begin{cases} \texttt{PNil} & : & \texttt{Prop (Program id)} & \equiv & \{v : \texttt{Program} \mid \texttt{prop}\ v = \texttt{Program id}\} & (\Gamma.1) \\ p_1 & : & \texttt{Prop (Program}\ s_1\texttt{)} & \equiv & \{v : \texttt{Program} \mid \texttt{prop}\ v = \texttt{Program}\ s_1\} & (\Gamma.2) \\ p_2 & : & \texttt{Prop (Program}\ s_2\texttt{)} & \equiv & \{v : \texttt{Program} \mid \texttt{prop}\ v = \texttt{Program}\ s_2\} & (\Gamma.3) \end{cases}$$

Next comes the type checking of the case expression that needs to check the body of each case branch under the environment extended with the pattern matching information.

In the `PNil` case, the type of the matching expression p_2, *i.e.,* the binding $(\Gamma.3)$, gets strengthened with the refinement of `PNil` (as we will formalize in the typing rule for T-Case in § 3):

$$\Gamma_{\texttt{PNil}} \supseteq \begin{cases} \texttt{PNil} & : & \{v : \texttt{Program} \mid \texttt{prop}\ v = \texttt{Program id}\} & (\Gamma_{\texttt{PNil}}.1) \\ p_1 & : & \{v : \texttt{Program} \mid \texttt{prop}\ v = \texttt{Program}\ s_1\} & (\Gamma_{\texttt{PNil}}.2) \\ p_2 & : & \{v : \texttt{Program} \mid \texttt{prop}\ v = \texttt{Program}\ s_2 \wedge \texttt{prop}\ v = \texttt{Program id}\} & (\Gamma_{\texttt{PNil}}.3) \end{cases}$$

Under this $\Gamma_{\texttt{PNil}}$ environment the goal is to prove that the expression returned by the `PNil` branch has the refinement type of the result, *i.e.,* that `p1` is a program with semantics $s_2\ .\ s_1$:

$$\Gamma_{\texttt{PNil}} \vdash p_1 : \{v : \texttt{Program} \mid \texttt{prop}\ v = \texttt{Program}\ (s_2\ .\ s_1)\} \qquad (GOAL)$$

Since the type of the variable p_1 as bound in the typing environment does not coincide with the required type, the type checker will check if the known type of p_1 is a subtype of the required type:

$$\Gamma_{\texttt{PNil}} \vdash \{v : \texttt{Program} \mid \texttt{prop}\ v = \texttt{Program}\ s_1\} \preceq \{v : \texttt{Program} \mid \texttt{prop}\ v = \texttt{Program}\ (s_2\ .\ s_1)\} \quad (S)$$

Essentially, this subtyping holds because under the environment $\Gamma_{\texttt{PNil}}$ the two types are equivalent. By the binding $(\Gamma_{\texttt{PNil}}.3)$ we can deduce that s_2 is the identity function, s_1 composed with the identity function is s_1, and, by the binding $(\Gamma_{\texttt{PNil}}.2)$, `p1` has the semantics of s_1. Yet, this reasoning involves the interaction of unification and higher-order functions, and as such is outside of the capabilities of a refinement type checker that is based on a first-order logic solver.

2.3 The PLEX Algorithm

Here, we introduce the PLEX solver, by demonstrating how it can be used to solve the (S) goal from the previous section. PLEX essentially extends the PLE algorithm used by `Liquid Haskell`, by enabling both higher-order reasoning (§ 2.3.2) and unification (§ 2.3.4). Furthermore, it extends the existing methods for function unfolding (§ 2.3.3) and generation of verification conditions (§ 2.3.1). These ingredients enable the resolution of the example goal from the previous section, which was previously beyond the capabilities of the vanilla PLE in `Liquid Haskell`.

2.3.1 Step 0: Generation of the Verification Condition. A refinement type checker reduces subtyping into logical reasoning by generating verification conditions (VCs) that are checked by an automated solver. To do this reduction it embeds each environment binding of the form $x : \{v : B \mid p\}$ into the logical formula $p[x/v]$ and ignores all the higher-order bindings. The VC checks if the embeddings of the typing environment and the left hand side of the subtyping imply the refinement of the right

hand side of the subtyping. Concretely, the subtyping goal (*S*) is reduced to the following VC:

$$\left.\begin{array}{lllll} & \texttt{prop PNil} & = & \texttt{Program id} & (1) \\ \wedge & \texttt{prop}\ p_1 & = & \texttt{Program}\ s_1 & (2) \\ \wedge & \texttt{prop}\ p_2 & = & \texttt{Program}\ s_2 & (3a) \\ \wedge & \texttt{prop}\ p_2 & = & \texttt{Program id} & (3b) \\ \hline \Rightarrow & \texttt{prop}\ p_1 & = & \texttt{Program}\ (s_2\ .\ s_1) & (GOAL) \end{array}\right\} (VC)$$

Such verification conditions are checked to be valid using an automated solver. In cases where the logic involved is decidable—*e.g.,* in queries that involve quantifier free formulas, uninterpreted functions, and linear arithmetic, such as the constraints on list lengths in the `minStackSize` function—an SMT solver can automatically prove the VCs. In the case of this (*VC*) however, the logic involved is higher-order. The functions `id`, s_1, and s_2 are represented as first-order variables after defunctionalization. A first-order solver can prove that s_2 is the identity function, because it knows that the data constructor `Program` is injective, but this alone is not sufficient to prove the goal.

2.3.2 Step 1: η and β Equivalences. Our first attempt to increase the precision of the solver, and ultimately prove the (*GOAL*), is to extend the reasoning with η and β equivalences. We denote with $\bowtie$ the equalities that are generated by our solver. Thus, $t_1 \bowtie t_2$ means that our solver has deduced that t_1 is equal to t_2. In the first-order logic encoding, $\bowtie$ reduces to SMT equality.

Our solver will scan the VC for partially applied functions. For each partially applied function, such as s_1 in the (*VC*), it will introduce one equality between the function and its η-expansion.

To encode the η-expansion, we need to encode lambdas in the VC. The encoding of lambdas can be either typed or untyped. Since the goal of PLEX is to approximate the evaluation of lambdas, the encoding needs to keep track of the functions' preconditions to ensure that they are not violated (as detailed in § 4.1). For example, assume a lambda abstraction that returns the previous of a positive number: $\lambda x\!:\!\{v\!:\!\mathsf{Int} \mid 0 < v\}.x - 1$. This lambda abstraction comes with the standard β, η equivalences that PLEX aims to approximate. Yet, the approximation algorithm (fig. 7) will only expand a lambda abstraction when its preconditions are not violated. We introduce the syntax $\lambda x\!:\!\tau_x.t$ to represent lambda abstractions in the VCs that are "guarded" by the preconditions in τ_x. For example, PLEX will only introduce the equality $(\lambda x\!:\!\{v\!:\!\mathsf{Int} \mid 0 < v\}.x - 1)\ t \bowtie t - 1$ when it can show $0 < t$. When ultimately we send the VC to a first-order solver, $\lambda x\!:\!\tau_x.t$ will be encoded as $\texttt{lam}\ (t[i_0/x])$, where the type is discarded, i_j is a set of predefined variables representing de Bruijn indices, and `lam` is an uninterpreted function of the proper sort. So, $\lambda x\!:\!\{v\!:\!\mathsf{Int} \mid 0 < v\}.x - 1$ will be encoded as $\texttt{lam\_int}\ (i_0^{\mathsf{Int}} - 1)$, where `lam_int` is an uninterpreted function that expects an integer argument and returns an integer and i_0^{Int} is an integer constant, defined for each SMT sort *e.g.,* boolean and data type. Dually, in the VCs we use the standard application syntax $t_1\ t_2$ to apply functions and potentially lambda abstractions, while in the logic we encode it with the uninterpreted function $\texttt{app}\ t_1\ t_2$. *Note* that with this de Bruijn, SMT encoding, we take α-equivalence "for free", *i.e.,* without the need for explicit reasoning in the PLEX solver.

Using the PLEX lambdas, we η-expand s_1 and the composition function, so the (*VC*) is extended with two new equalities: (For readability, we write $(s_2\ .\ s_1)$ instead of $(.\ s_2\ s_1)$.)

$$\left.\begin{array}{lllll} & & \cdots & & \\ \wedge & s_1 & \bowtie & \lambda x_4\!:\!Stack.s_1\ x_4 & (4) \\ \wedge & s_2\ .\ s_1 & \bowtie & \lambda x_5\!:\!Stack.((s_2\ .\ s_1)\ x_5) & (5) \\ \hline \Rightarrow & \texttt{prop}\ p_1 & = & \texttt{Program}\ (s_2\ .\ s_1) & (GOAL) \end{array}\right\} (VC1)$$

This simulation of η-expansion is a critical step in reasoning about functions in two respects. First, it converts functions to their fully applied form, thus enabling reasoning about them, especially when coexisting with the β-reduction. Even though the β-reduction is not used in this example,

our solver also simulates it by introducing equalities between functions and their applications, as formalized in § 4. Second, η- and β-equivalences are elevated to theorems in the overall refinement type system, eliminating the need to postulate them as axioms. This avoids the complications typically arising from their interaction with subtyping, as discussed in Vazou and Greenberg [2022], where users have no control over the types of newly introduced bound variables.

In general, the η-reduction rule can be applied infinitely many times. Yet, our solver is designed to carefully select to expand only the largest application chain, so that termination is guaranteed.

2.3.3 Step 2: Function Unfolding. The next step is to unfold the function definitions. The vanilla PLE already equates the fully applied functions that are reflected into the logic, with their unfolded reduct. Assuming that function composition is reflected, the solver can unfold $(s_2\ .\ s_1)\ x_5$ to $s_2\ (s_1\ x_5)$:

$$\left.\begin{array}{llll} & & \ldots & \\ \wedge & (s_2\ .\ s_1)\ x_5 & \bowtie & s_2\ (s_1\ x_5) \\ \hline \Rightarrow & \texttt{prop}\ p_1 & = & \texttt{Program}\ (s_2\ .\ s_1) \end{array}\right. \quad \begin{array}{r} (6) \\ (GOAL) \end{array} \Big\} (VC2)$$

Note that even though the reflection and PLE techniques of Vazou et al. [2017] do not reason about partially applied functions, they provide a solid foundation for higher-order reasoning, and in this work we show that they can coexist with the new reductions that PLEX adds.

2.3.4 Step 3: Unification. The next step we take is to replace s_2 with `id` in the equality (6).

$$\left.\begin{array}{llll} & & \ldots & \\ \wedge & s_2\ (s_1\ x_5) & \bowtie & \texttt{id}\ (s_1\ x_5) \\ \hline \Rightarrow & \texttt{prop}\ p_1 & = & \texttt{Program}\ (s_2\ .\ s_1) \end{array}\right. \quad \begin{array}{r} (7) \\ (GOAL) \end{array} \Big\} (VC3)$$

The fact that s_2 is equal to `id` can be deduced by a first-order logic solver, given the equalities (3*a*) and (3*b*). Of course, extending a VC with new equalities that substitute f with g, for all f and g that the solver can deduce equal, is not efficient. But, the equality $s_2 = \texttt{id}$ is special because it is only true inside the branch and the unifying equality can be locally decided by the strengthened refinement of the matching expression. In this light, we developed an approximation of the unification procedure that selects the potential unifiers when type checking the case expressions. In § 3.3 we provide a formalization of the unification procedure and we prove that it is both terminating and sound.

In our example, the unification will select potential unifiers from the case strengthened refinement `{v:Program|prop v = Program s2 ∧ prop v = Program id}` and will generate the unifier $s_2 \cong \texttt{id}$. Thus, our solver will create one additional equality at each position that s_2 is applied.

2.3.5 Completing the Proof. In the final step the function unfolding will expand the definition of the, now fully applied, identity function, leading to the following goal:

$$\begin{array}{llll} & \texttt{prop PNil} & = & \texttt{Program id} & (1) \\ \wedge & \texttt{prop}\ p_1 & = & \texttt{Program}\ s_1 & (2) \\ \wedge & \texttt{prop}\ p_2 & = & \texttt{Program}\ s_2 & (3a) \\ \wedge & \texttt{prop}\ p_2 & = & \texttt{Program id} & (3b) \\ \wedge & s_1 & \bowtie & \lambda x_4 : Stack.s_1\ x_4 & (4) \\ \wedge & s_2\ .\ s_1 & \bowtie & \lambda x_5 : Stack.(s_2\ .\ s_1)\ x_5 & (5) \\ \wedge & (s_2\ .\ s_1)\ x_5 & \bowtie & s_2\ (s_1\ x_5) & (6) \\ \wedge & s_2\ (s_1\ x_5) & \bowtie & \texttt{id}\ (s_1\ x_5) & (7) \\ \wedge & \texttt{id}\ (s_1\ x_5) & \bowtie & s_1\ x_5 & (8) \\ \hline \Rightarrow & \texttt{prop}\ p_1 & = & \texttt{Program}\ (s_2\ .\ s_1) & (GOAL) \end{array}$$

Following the equalities (5) to (8), we can deduce that $s_2\ .\ s_1 \bowtie \lambda x_5 : Stack.s_1\ x_5$ and thus, by (4) and α-equivalence, that is equal to s_1. Thus, `Program` $(s_2\ .\ s_1)$ is equal to `Program` s_1, which by (2) is equal to `prop` p_1. This completely proves the goal.

From this chain of, not so trivial, equational reasoning, we can see the need of having an SMT solver to finally prove the goal. A similar reasoning is applied in the inductive case of the `compose` function (of § 2.2.1), where the solver will have to reason about the composition of two functions, and will have to apply the η and β-reductions, unfolding, and unification to prove the desired goal.

2.4 Arithmetic for Free: Completing the Compiler

Even though PLEX is designed to reason about higher-order functions, it is ultimately using the SMT solver to prove the goals. After properly encoding higher-order functions, PLEX can enjoy the strengths of SMT and, in particular, their substantial automation on resolving arithmetic goals.

To illustrate this, we will complete the compiler of the expression language into the stack machine language, and prove its correctness. The correctness is guaranteed by the type signature of `compile`, in which we assert that the semantics of the compiled program must be equal to pushing on top of the stack the evaluation of the given expression.

```
{-@ compile :: e:Expr → Prop (Program (push $ eval e)) @-}
compile :: Expr → Program
compile (EConst x)   = PCons (OpPush x) id PNil
compile (EAdd e1 e2) = PCons OpAdd ((push $ eval e1) . (push $ eval e2))
    (compose (push $ eval e2) (push $ eval e1) (compile e2) (compile e1))
compile (EMul e1 e2) = PCons OpMul ((push $ eval e1) . (push $ eval e2))
    (compose (push $ eval e2) (push $ eval e1) (compile e2) (compile e1))
compile (ENeg e)     = PCons OpMul ((push $ -1) . (push $ eval e))
    (PCons (OpPush $ -1) (push $ eval e) (compile e))
```

Here, as in the `compose` example, the higher-order reasoning of PLEX is crucial to prove the correctness of the specification. However, the arithmetic reasoning is left to the SMT solver. For instance, in the `ENeg` case, the SMT easily proves that `-a = a * (-1)`. In traditional proof assistants, like Agda, such arithmetic reasoning would require the user to provide the missing lemmas explicitly. The same holds if we decide to push the two operands of `EAdd` and `EMul` in a different order than the one defined in `eval`—this is only *semantically* correct as these operations are commutative. We further compare our proofs to their counterparts in Agda in § 5.4.

3 Refinement Types with Dependent Pattern Matching

In this section we present λ^R, a refinement typed lambda calculus with dependent pattern matching. We start, in § 3.1, by presenting the syntax and semantics of λ^R, a standard refined calculus, but with a more delicate definition of function equality that was required for the metatheory. Then, § 3.2 presents the typing rules of λ^R, that accumulate the equalities obtained by pattern matching. Next, § 3.3 presents the unification algorithm required for the dependent pattern matching. Finally, § 3.4 presents the metatheory of λ^R. (The proofs of all the theorems are in the supplementary material.)

3.1 Core Calculus λ^R

The calculus λ^R is a refinement typed lambda calculus with data types and reflected definitions, similar to [Vazou et al. 2017]. We start by presenting the syntax of the calculus (§ 3.1.1) and then define its operational (§ 3.1.2) and denotational (§ 3.1.3) semantics.

3.1.1 Syntax. Figure 1 presents the syntax of λ^R.

Operators of λ^R ($\odot$) include the standard boolean ($\neg$ and $\wedge$), integer ($+$ and $-$), and (in)equality ($=$ and $<$) operators. *Constants* (c) include the operators, as well as the constants for boolean and integer

$$
\begin{array}{rrcl}
\textbf{Operators} & \odot & ::= & \neg \mid \wedge \mid + \mid - \mid = \mid < \mid \ldots \\
\textbf{Constants} & c & ::= & \odot \mid \mathsf{true} \mid \mathsf{false} \mid 0, -1, 1, \ldots \\
\textbf{Values} & w & ::= & c \mid \lambda x{:}\tau.e \mid D\ \overline{w} \mid \mathtt{Undef} \\
\textbf{Expressions} & e, r & ::= & w \mid x \mid e\ e \\
 & & \mid & \mathsf{case}\ x = e\ \mathsf{of}\{\overline{D\ \overline{x} \rightarrow e}\} \\
\textbf{Bindings} & b & ::= & e \mid \mathsf{let\ rec}\ x{:}\tau\ =\ e\ \mathsf{in}\ b \\
\textbf{Programs} & p & ::= & b \mid \mathsf{reflect}\ x{:}\tau\ =\ e\ \mathsf{in}\ p \\
\textbf{Predicates} & r & ::= & \{e \mid \exists \Gamma.\Gamma \vdash_{\mathrm{STLC}} e{:}\mathsf{Bool}\} \\
\textbf{Base Types} & B & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{T} \\
\textbf{Refined Types} & \tau & ::= & \{x{:}B[^{\Downarrow}] \mid r\} \mid x{:}\tau \rightarrow \tau
\end{array}
$$

Typing Environments

$$
\begin{array}{rcl}
\Gamma & ::= & \emptyset \mid \Gamma, x{:}\tau \\
R & ::= & \emptyset \mid R, x{:}\tau := e \\
U & ::= & \emptyset \mid U, x \cong e
\end{array}
$$

Evaluation Context

$$
\begin{array}{rcl}
C & ::= & \bullet \mid C\ e \mid c\ C \\
 & \mid & D\ \overline{w}\ C\ \overline{e} \\
 & \mid & \mathsf{case}\ x = C\ \mathsf{of}\{\overline{D\ \overline{y} \rightarrow e}\}
\end{array}
$$

Fig. 1. Syntax of the Source Calculus λ^R.

$$
\frac{}{C[p] \hookrightarrow C[p'],\quad \text{if } p \hookrightarrow p'} \qquad c\ w \hookrightarrow \delta(c, w)
$$

$$
\mathsf{case}\ y = D_j\ \overline{e}\ \mathsf{of}\{\overline{D_i\ \overline{x_i} \rightarrow e_i}\} \hookrightarrow e_j\ [D_j\ \overline{e}/y][\overline{e}/\overline{x_i}] \qquad (\lambda x{:}\tau.e)\ e_x \hookrightarrow e\ [e_x/x]
$$

$$
\mathsf{let\ rec}\ x : \tau\ =\ e\ \mathsf{in}\ b \hookrightarrow b\ [\mathsf{fix}\ (\lambda x{:}\tau_x.e)/x] \qquad \mathsf{fix}\ p \hookrightarrow p\ (\mathsf{fix}\ p)
$$

$$
\mathsf{reflect}\ x : \tau\ =\ e\ \mathsf{in}\ p \hookrightarrow p\ [\mathsf{fix}\ (\lambda x{:}\tau_x.e)/x]
$$

Fig. 2. Operational Semantics of λ^R: Definition of small step, reduction relation $p \hookrightarrow p$.

values. *Values* (w) include the constants, typed lambda abstractions ($\lambda x : \tau.e$), data constructors applied to values ($D\ \overline{w}$), as well as the untyped, intermediate value `Undef` that cannot be written by the user, but is required for the metatheoretical development.

Expressions of λ^R (e or r) include values, variables (x, but we also use f for variables of functions), function applications $e\ e$, and case expressions $\mathsf{case}\ x = e\ \mathsf{of}\{D\ \overline{x} \rightarrow e\}$. *Bindings* ($b$) include expressions and potentially recursive let bindings. Finally, *programs* (p) include bindings and reflected programs [Vazou et al. 2017], *i.e.,* programs whose definition is reflected in the logic.

Types are separated into *base types* (B) and *refined types* (τ). Base types include integers (Int), booleans (Bool), and type constructors (T), that arise from data type declarations. Predicates (r) are boolean expressions that are used to introduce *refined types*. Each predicate must be typed as boolean using simple type lambda calculus typing, rather than the rules that we will see in § 3.2. There are two forms of refined types: base types and function types. Base types are refined with a boolean expression $\{x{:}B \mid r\}$, that includes all the expressions e of type B for which the expression $r[e/x]$ evaluates to true. Optionally, we annotate the base type with a downarrow, *e.g.,* $\{x{:}B^{\Downarrow} \mid r\}$, to type terminating expressions of type B that satisfy the boolean expression r. The function type $x{:}\tau_x \rightarrow \tau$ binds the name x with the type τ_x that can appear in the refinement of the result type τ.

Notation: For brevity, we abbreviate $\{x : B \mid \mathsf{true}\}$ as B and $\{x : B^{\Downarrow} \mid \mathsf{true}\}$ as $B^{\Downarrow}$. We write $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ for the expression $\mathsf{case}\ \_ = e\ \mathsf{of}\{\mathsf{true} \rightarrow e_1; \mathsf{false} \rightarrow e_2\}$.

3.1.2 Operational Semantics. Figure 2 presents the call-by-name, small step, contextual operational semantics of λ^R. The evaluation context fixes the order of evaluation. The small step semantics itself is given by the relation $p \hookrightarrow p'$, which holds when the program p evaluates to the program p' in one step. Most of the rules are fairly standard; the rules use an explicit fixpoint operator to handle recursive definitions as well as reflected functions. That is, the reflect annotation has no runtime effect and is used only for typing purposes.

To evaluate applications of primitive functions, we use the δ function that essentially performs the operation. For example, $\delta(+, 30)$ returns the curried constant $+_{30}$ and $\delta(+_{30}, 12)$ returns the result 42. For simplicity, we write $\delta(\odot, c_1, c_2)$ to denote this evaluation in two steps. For example, $\delta(+, 30, 12)$ returns 42. Finally, we define $\hookrightarrow^\star$ to be the reflexive, transitive closure of $\hookrightarrow$.

Equality Operator. The definition of the equality operator is delicate. Since λ^R has four kinds of values, the behavior of $\delta(=, w_1, w_2)$ depends on the shapes of w_1 and w_2. `Undef` is equal only to itself, while equality for constants and data constructors is straightforward.

Comparing lambda abstractions is more complicated. Ideally, we would like to compare two functional values $f \doteq \lambda x{:}\tau_1.e_1$ and $g \doteq \lambda y{:}\tau_2.e_2$ extensionally, meaning that $f = g$ *if and only if* for every expression e_a, the equality $e_1[e_a/x] = e_2[e_a/y]$ holds. However, in the context of refinement types the challenge is that the domains τ_1 and τ_2 might not be the same and still we would like to compare these functions. For example, the domain of f could be $\tau_1 \doteq \{x{:}\mathsf{Int} \mid 0 < x\}$, *i.e.,* accepting only positive integers, and of g could be $\tau_2 \doteq \{y{:}\mathsf{Int} \mid \mathsf{false}\}$, *i.e., g* should not be called, and its body could violate the refinement specifications or even diverge! Thus, the crux of the matter is on which domain should two functions be compared.

Neither the intersection nor the union of the domains is a good choice. If we compared f and g on the intersection of their domains, that would be false $\wedge\ 0 < x$, thus false, and the two functions would be decided equal on the empty domain, leading to an inconsistent system [Vazou and Greenberg 2022]. If we compared them on the union of their domains, that would be false $\vee\ 0 < x$, thus $0 < x$, permitting to call g on any positive integer; thus violating its specification.

To address this, we make use of the special value `Undef`, to explicitly encode that the function is not defined. Concretely, we define the $\textsc{Coerce}(\tau, e)$ function, that, similar to [Belo et al. 2011; Knowles and Flanagan 2010; Siek et al. 2021], coerces an expression e to the type τ, by inserting the `Undef` value when the type is not satisfied.

$$\textsc{Coerce}(\{x : B \mid r\}, e) \doteq \mathsf{if}\ r[e/x]\ \mathsf{then}\ e\ \mathsf{else}\ \mathtt{Undef}$$
$$\textsc{Coerce}(x{:}\tau_x \to \tau, e) \doteq \lambda y{:}\tau_x.\mathsf{if}\ \textsc{Coerce}(\tau_x, y) = \mathtt{Undef}\ \mathsf{then}\ \mathtt{Undef}\ \mathsf{else}\ \textsc{Coerce}(\tau, e\ y)$$

For the base case, we just check if the expression satisfies the refinement. For the function case, we η-expand. If the coercion of the provided argument is `Undef`, then the coercion returns `Undef`, otherwise it coerces the body. Note that this definition is carefully designed to properly propagate the `Undef` value and ensure that a potential `Undef` of the coercion of the argument cannot interact with the body of the function. With `Undef`, functions are turned total, *i.e.,* defined for all inputs, but they can still diverge. For the purposes of equality checking, we say that a function can diverge, when there exists a value in its domain that makes the body diverge. Formally:

$$\mathsf{Div}(\lambda x{:}\tau.e) \doteq \exists e_x{:}\mathtt{Erase}(\tau).\ e[e_x/x] \not\hookrightarrow^* w$$
$$\mathtt{Erase}(\{x{:}B \mid r\}) \doteq B \qquad \mathtt{Erase}(x{:}\tau_x \to \tau) \doteq \mathtt{Erase}(\tau_x) \to \mathtt{Erase}(\tau)$$

where $\mathtt{Erase}(\tau)$ is the STLC type of τ, *i.e.,* after erasing the refinements, and $e : \tau$ checks the STLC type of e, after the refinements on the lambdas are erased. We write $\exists e_x : \mathtt{Erase}(\tau)$ as an abbreviation for $\exists e_x$ *such that* $\emptyset \vdash_{\mathrm{STLC}} e_x{:}\mathtt{Erase}(\tau)$.

We can now define the equality of two lambdas. Let $f_1 \doteq \lambda x{:}\tau_1.e_1$ and $f_2 \doteq \lambda y{:}\tau_2.e_2$. Further, let cf_1^τ be the coercion of f_1 to some result type τ, *i.e.,* $cf_1^\tau \doteq \textsc{Coerce}(x{:}\tau_1 \to \tau, f_1)$ and similarly, $cf_2^\tau \doteq \textsc{Coerce}(y{:}\tau_2 \to \tau, f_2)$ Then, equality is defined as:

$$\delta(=, f_1, f_2) \doteq \begin{cases} \mathtt{Undef}, & \textit{if}\ \mathsf{Div}(f_1)\ \textit{or}\ \mathsf{Div}(f_2) \\ \forall e_x{:}\mathtt{Erase}(\tau_1).\exists \tau.cf_1^\tau\ e_x = cf_2^\tau\ e_x \hookrightarrow^* \mathsf{true}, & \textit{otherwise} \end{cases}$$

Equality is undefined if either function may diverge; otherwise, it checks the equality of the coerced functions on all values in the erased domain. Two functions are equal *if and only if* they are defined

and equal on their domains, since `Undef` equals only itself. This definition is non-computable and cannot appear outside refinement propositions, where it is not evaluated but approximated by the type checker. In [Vazou et al. 2017], extensional equality was sufficient, because the PLE symbolic evaluation did not aim to generate function equalities. In contrast, our development needs to ensure that function specifications are respected, even during the symbolic evaluation.

3.1.3 Denotational Semantics. We use the evaluation semantics to define the denotational semantics of the types. We define the function $[\![\tau]\!]$ that returns the set of programs that have type τ:

$$\begin{array}{rcl} [\![\{x{:}B \mid r\}]\!] & \doteq & \{e \mid \emptyset \vdash_{\text{STLC}} e{:}B \wedge\ r[e/x] \hookrightarrow^{\star} \text{true}\} \\ [\![\{x{:}B^{\Downarrow} \mid r\}]\!] & \doteq & [\![\{x{:}B \mid r\}]\!] \cap \{e \mid \exists w.\ e \hookrightarrow^{\star} w\} \\ [\![x{:}\tau_x \to \tau]\!] & \doteq & \{e \mid \forall e_x \in [\![\tau_x]\!].\ (e\ e_x) \in [\![\tau[e_x/x]]\!]\} \end{array}$$

The denotation of a refined type $\{x : B \mid r\}$ is the set of programs e that have type B using the standard STLC typing rules and for which $r[e/x]$ evaluates to true and $\{x{:}B^{\Downarrow} \mid r\}$ is a restriction on the previous denotation that only includes terminating programs. Finally, the denotation of a function type $x : \tau_x \to \tau$ is the set of programs e such that for whatever expression e_x in the denotation of τ_x, the application $e\ e_x$ is in the denotation of τ in which x is substituted by e_x.

Closing Substitutions. We extend the denotation of types to typing environments. A typing environment Γ binds variables to types, *i.e.,* $x_1 : \tau_1, \ldots, x_n : \tau_n$. The closing substitution of a typing environment Γ is the set of all substitutions, *i.e.,* mappings from variables to values $x_1 \mapsto w_1, \ldots, x_n \mapsto w_n$, where each x_i is bound to a value w_i in the denotation of the type $\Gamma(x_i)$.

$$[\![\Gamma]\!] \doteq \{\theta \mid \forall x{:}\tau \in \Gamma, \theta(x) \in [\![\tau]\!]\}$$

3.2 Refinement Type Checking

Figure 3 presents the typing rules for λ^R. Essentially, the rules are similar to the rules from the literature [Vazou et al. 2017], where a reflection environment R is used to track the reflected functions, but extended with a unification environment U to track the equalities obtained by pattern matching. Intuitively, the typing judgment $\Gamma; R; U \vdash p : \tau$ states that the program p has type τ under the environments Γ, R, and U. As standard in refinement type systems, it uses two additional judgments: the first judgment $\Gamma \vdash \tau$ states that the type τ is well-formed under the typing environment Γ, *i.e.,* all the free variables in τ are bound in Γ and all the refinements are boolean expressions; the second judgment, $\Gamma; R; U \vdash \tau_1 \preceq \tau_2$, states that the type τ_1 is a subtype of τ_2, *i.e.,* any expression with type τ_1 can be used in a context that expects type τ_2.

Environments. The *reflection environment* R is a sequence of reflected function definitions $f_1 : \tau_1 := e_1, \ldots, f_m : \tau_m := e_m$ where f_i are distinct identifiers and e_i are their reflected definitions. Finally, the *unification environment* U is a sequence of equalities $x_1 \cong e_1, \ldots, x_k \cong e_k$, where x_i are distinct identifiers and e_i are expressions in λ^R.

Typing Rules. Figure 3 defines the typing judgment $\Gamma; R; U \vdash p : \tau$, that using the three environments Γ, R, and U, states that the program p has type τ. Rule T-Var looks up the type of a variable in the typing environment. T-Con uses the function $\mathsf{primType}(c)$ to decide the type of constants. For precise typing, this function should return the exact type for constants.

T-Sub is using subtyping to weaken the type of an expression, while T-Exact is using selfification [Knowles and Flanagan 2010] to strengthen the type of an expression with its exact definition. T-Fun types lambda abstractions by typing the body under a typing environment extended with the argument type. Dually, T-App types a function application $e\ e_x$ by ensuring that the argument has the required type. Note that the type of the argument e_x can be implicitly weakened by the

Typing $\boxed{\Gamma; R; U \vdash p : \tau}$

$$\frac{x : \tau \in \Gamma}{\Gamma; R; U \vdash x : \tau}\ \text{T-Var} \qquad \frac{}{\Gamma; R; U \vdash c : \mathsf{primType}(c)}\ \text{T-Con}$$

$$\frac{\Gamma; R; U \vdash e : \tau_1 \qquad \Gamma; R; U \vdash \tau_1 \preceq \tau_2}{\Gamma; R; U \vdash e : \tau_2}\ \text{T-Sub} \qquad \frac{\Gamma; R; U \vdash e : \{x : B \mid r\}}{\Gamma; R; U \vdash e : \{x : B \mid r \wedge x = e\}}\ \text{T-Exact}$$

$$\frac{\Gamma, x : \tau_x; R; U \vdash e : \tau}{\Gamma; R; U \vdash \lambda x : \tau_x . e : (x : \tau_x \to \tau)}\ \text{T-Fun} \qquad \frac{\Gamma; R; U \vdash e : (x : \tau_x \to \tau) \qquad \Gamma; R; U \vdash e_x : \tau_x}{\Gamma; R; U \vdash e\ e_x : \tau[e_x/x]}\ \text{T-App}$$

$$\frac{\Gamma, f : \tau_f; R; U \vdash e_f : \tau_f \quad \Gamma, f : \tau_f \vdash \tau_f \quad \Gamma, f : \tau_f; R; U \vdash b : \tau \quad \Gamma \vdash \tau \quad \mathsf{TCheck}(\mathsf{rec}\ f : \tau_f = e_f)}{\Gamma; R; U \vdash \mathsf{let\ rec}\ f : \tau_f\ =\ e_f\ \mathsf{in}\ b : \tau}\ \text{T-Rec}$$

$$\frac{\tau_R \doteq \mathsf{reflType}(\tau_f, e) \qquad \Gamma; R, f : \tau_R := e; U \vdash \mathsf{let\ rec}\ f : \tau_R\ =\ e\ \mathsf{in}\ p : \tau}{\Gamma; R; U \vdash \mathsf{reflect}\ f : \tau_f\ =\ e\ \mathsf{in}\ p : \tau}\ \text{T-Reflect}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau \qquad \forall i. \mathsf{primType}(D_i) = \overline{y_j : \tau_j} \to \{x_i : \mathsf{T} \mid r_{xi}\} \qquad \Gamma; R; U \vdash e : \{v : \mathsf{T} \mid r_v\} \\ \Gamma, \overline{y_j : \tau_j}, x : \{v : \mathsf{T} \mid r_v \wedge r_{xi}\}; R; U, \mathsf{Unify}(\mathsf{dom}(\Gamma) \setminus \mathsf{dom}(R), r_v \wedge r_{xi}) \vdash e_i : \tau\end{array}}{\Gamma; R; U \vdash \mathsf{case}\ x = e\ \mathsf{of}\{D_i\ \overline{y_j} \to e_i\} : \tau}\ \text{T-Case}$$

Well formedness $\boxed{\Gamma \vdash \tau}$

$$\frac{\Gamma, x : B; \emptyset; \emptyset \vdash r : \mathsf{Bool}^{\Downarrow}}{\Gamma \vdash \{x : B \mid r\}}\ \text{WF-Base} \qquad \frac{\Gamma \vdash \tau_x \qquad \Gamma, x : \tau_x \vdash \tau}{\Gamma \vdash x : \tau_x \to \tau}\ \text{WF-Fun}$$

Subtyping $\boxed{\Gamma; R; U \vdash \tau \preceq \tau}$

$$\frac{\Gamma; R; U \vdash \{x : B \mid r_1\} \Rightarrow \{x : B \mid r_2\}}{\Gamma; R; U \vdash \{x : B \mid r_1\} \preceq \{x : B \mid r_2\}}\ \preceq\text{-Base} \qquad \frac{\begin{array}{c}\Gamma; R; U \vdash \tau_{2x} \preceq \tau_{1x} \\ \Gamma, x : \tau_{2x}; R; U \vdash \tau_1 \preceq \tau_2\end{array}}{\Gamma; R; U \vdash x : \tau_{1x} \to \tau_1 \preceq x : \tau_{2x} \to \tau_2}\ \preceq\text{-Fun}$$

Fig. 3. Typing rules for λ^R.

subtyping rule T-Sub. Further, the typing rule substitutes the free occurrences of x with e_x in the function body e, using the capture avoiding substitution $[e_x/x]$.

The T-Rec types recursive definitions of a function f with declared type τ_f, by typing its body under an extended typing environment where f is bound to τ_f. The rest of the program is also typed under the same extended environment. The rule also checks well formedness of both the type τ_f and the resulting type τ: the type τ_f should be checked to be well-formed since it is provided by the user, while well formedness of τ ensures that the declared function f does not appear in τ. The premise TCheck checks that the recursive definition complies with its termination annotation and can be implemented with various alternative techniques [Barthe et al. 2004; Jones and Bohr 2004; Sereni and Jones 2005]; termination checking is not the focus of this paper.

The rule T-Reflect types the reflected program reflect $f : \tau_f\ =\ e$ in p in three steps. First, it extends the reflection environment to capture f's definition. Second, it reflects the declaration's

$$\frac{\forall \theta \in [[\Gamma]].\ [[\theta \cdot \{x : B \mid r_1\}]] \subseteq [[\theta \cdot \{x : B \mid r_2\}]]}{\Gamma; R; U \vdash \{x{:}B \mid r_1\} \Rightarrow \{x{:}B \mid r_2\}}\ \textsc{I-Den}$$

Fig. 4. Denotational, Undecidable Entailment Checking.

type to precisely capture its definition. Finally, it converts the definition into a recursive one and checks it against the refined type and extended environment. The type reflection is performed by $\mathsf{reflType}(\tau_f, e)$, which strengthens τ_f with the exact definition e:

$$\begin{array}{lcl} \mathsf{reflType}(\{x{:}B \mid r\}, e) & \doteq & \{x{:}B^{\Downarrow} \mid r \wedge x = e\} \\ \mathsf{reflType}(x{:}\tau_x \rightarrow \tau, \lambda x{:}\tau_x.e) & \doteq & x{:}\tau_x \rightarrow \mathsf{reflType}(\tau, e) \end{array}$$

To ensure consistency of the resulting system, only terminating expressions can be reflected. Thus, the result type of the reflected definition is annotated with $^{\Downarrow}$.

Finally, the rule T-Case types a case expression case $x = e$ of$\{\overline{D_i\ \overline{y} \rightarrow e_i}\}$. Each alternative e_i is checked in a typing environment where the type of x is strengthened with the refinement of the matching expression e, *i.e.,* r_v, as well as the refinement of the corresponding constructor D_i, *i.e.,* r_{xi}. This strengthening ensures path sensitivity. *The novelty of our system* is the observation that the pattern matching information ($r_v \wedge r_{xi}$) is critical for the logical evaluation of the PLEX algorithm. Thus, the rule T-Case accumulates this information in the unification environment U, using the unification function $\mathsf{Unify}(\cdot, \cdot)$ that we present in § 3.3.

The vanilla case rule, *e.g.,* from [Vazou et al. 2017], performs pattern matching, *i.e.,* each alternative uses the information of the data constructor. Our extension also uses the information obtained by matching on the index of the type, thus enabling *dependent pattern matching*.

Example 3.1. To illustrate the T-Case rule, consider the expression from § 2.2.1 inside `compose`: case $x = p_2$ of$\{PNil \rightarrow e_n;\ PCons\ \ldots \rightarrow e_c\}$ which is type checked in an environment that contains the binding $p_2 : \{x : \texttt{Program} \mid \mathsf{prop}\ x = \texttt{Program}\ s_2\}$. The first premise ensures that the type of the expression is well-formed. The second and third premises ensure that p_2 is well-typed for some type `Program` and that all the data constructors used in pattern matching are valid patterns for `Program`. The last premise checks that all case bodies have the same type, where at each case the environment is strengthened by bringing into scope the arguments of the data constructor and by utilizing the refinement of the return type of the constructor. In the `PNil` case the type of the bound variable x is strengthened to $\{x : \texttt{Program} \mid \mathsf{prop}\ x = \texttt{Program}\ s_2 \wedge \mathsf{prop}\ x = \texttt{Program}\ \texttt{id}\}$, and the unification environment is extended with the information that $s_2 \cong \texttt{id}$, which is obtained through unification of $\mathsf{prop}\ x = \texttt{Program}\ s_2 \wedge \mathsf{prop}\ x = \texttt{Program}\ \texttt{id}$.

Well-formedness. The judgment $\Gamma \vdash \tau$ checks that the type τ is well-formed. The rules are standard, *i.e.,* the rule WF-Base checks that the refinement of each base type is a terminating boolean expression while the rule WF-Fun checks that the argument type is well-formed and the result type is well-formed under an extended typing environment.

Subtyping. The judgment $\Gamma; R; U \vdash \tau_1 \preceq \tau_2$ checks that the type τ_1 is a subtype of τ_2. The function rule $\preceq$-Fun is checking subtyping using contravariance for the argument type and covariance for the result type, in an environment extended with the strongest argument type. The base rule $\preceq$-Base simply reduces subtyping to entailment, that we describe next.

Entailment Checking. Figure 4 presents the rule I-Den that uses the type denotations to check entailment. Concretely, it checks that for all closing substitutions θ of the context Γ, the denotation of the refined type $\{x{:}B \mid r_1\}$ is a subset of the denotation of $\{x{:}B \mid r_2\}$. This implication checking

Unify	:	$(\mathcal{P}(x), e) \to U$
Unify(xs, e)	$\doteq$	elimCircles $\cdot$ concatMap (unifiers xs)
	$\cdot$	fix closure $\cdot$ candidates
	\$	e
candidates	:	$e \to \mathcal{P}(e, e)$
candidates$(e_1 \wedge e_2)$	$\doteq$	candidates$(e_1) \cup$ candidates(e_2)
candidates$(e_1 = e_2)$	$\doteq$	$\{(e_1, e_2)\}$
candidates(e)	$\doteq$	$\{\}$
unifiers	:	$\mathcal{P}(x) \to \mathcal{P}(e, e) \to U$
unifiers xs (e, e)	$\doteq$	$\{\}$
unifiers xs $(x, e) \mid x \in xs$	$\doteq$	$\{x \cong e\}$
unifiers xs $(e, x) \mid x \in xs$	$\doteq$	$\{x \cong e\}$
unifiers xs $(D\ e_{11} \ldots e_{1n}, D\ e_{21} \ldots e_{2n})$	$\doteq$	$\bigcup_{i \in 1 \ldots n}$ unifiers xs (e_{1i}, e_{2i})
unifiers xs $(\lambda x{:}\tau_1.e_1, \lambda y{:}\tau_2.e_2) \mid x \notin xs, y \notin xs$	$\doteq$	unifiers xs $(e_1[y/x], e_2)$
unifiers xs (e_1, e_2)	$\doteq$	$\{\}$
closure – transitive closure	:	$\mathcal{P}(e, e) \to \mathcal{P}(e, e)$
elimCircles – remove unification cirlces	:	$U \to U$

Fig. 5. The Unify algorithm. The reading of the definitions is sequential. The full definitions of closure and elimCircles are in the appendix.

is precise in that it cannot fail if the entailment is true, but is not decidable, since it requires checking all the possible substitutions. In § 4, we present a decidable approximation of this rule.

3.3 Unification Algorithm for Dependent Pattern Matching

Here we define the unification algorithm used in the case rule to build the unification environment.

Ideally, this unification procedure should return the most general unifier, *i.e.,* the set of all equalities that are implied by the input expression, as defined below.

Definition 3.2 (Complete Unifier). The complete unifier of some boolean expression r, namely $\mathcal{U}(r)$, is the biggest collection of pairs $x \cong e_x$ such that $r \Rightarrow x = e_x$.

Sadly, computing the complete unifier is undecidable: higher-order unification [Miller 1991] is known to be undecidable and can be reduced to computing the complete unifier. To do so, suppose we want to unify two higher-order expressions e_1 and e_2, if we could compute the complete unifier $\mathcal{U}(e_1 = e_2)$ then the result would be a solution of the original unification problem.

Approximation of the Complete Unifier. To preserve decidable type checking, we define an approximation of the complete unifier. Figure 5 shows the algorithm Unify that computes the unification environment, *i.e.,* the set of unifications that respect the typing environment. The algorithm takes as input the set of variables that can freely be used, *i.e.,* $\mathtt{dom}(\Gamma) \setminus \mathtt{dom}(R)$ in the T-Case rule, and the boolean expression that implies the unification. Then, it proceeds in four steps, dictated by the four main procedures: candidates, closure, unifiers, and elimCircles.

First, the algorithm computes the set of candidate unifications, *i.e.,* all the equalities that are syntactically present in subconjuctions of the input expression and then it computes their transitive closure. The procedure unifiers performs the unification between expressions. It operates similarly

to first-order unification. If one expression is a variable to be unified, *i.e.,* it belongs into the input variable set, then the algorithm unifies the variable with the expression. If both expressions are applications of the same data constructor, since they are injective the arguments must be equal, and the algorithm unifies them one by one. If both expressions are lambda abstractions, their bodies are unified, up to renaming. In all other cases, the algorithm returns no unification. At the end of the process, the elimCircles function is used to collect a subset of the unification that is not cyclic.

Our unification algorithm Unify does not do *pattern unification* [Miller 1991], *i.e.,* the reconstruction of a function definition from its application. For example, given the unification problem $f\ x = \mathsf{true}$, where both f and x are variables; with pattern unification we can deduce that $f \doteq \lambda y : \mathsf{Int}.\mathsf{true}$. However, this does not extend naturally to refinement types. In Miller [1991]'s type system, x does not have any semantic constraints so f has only one solution. But in a refinement type system, x can be constrained by its type. If, for example, x has the type $\{x : \mathsf{Int} \mid x \geq 10\}$, then f can have multiple solutions, for example, $\lambda x : \mathsf{Int}.x \geq 10$, or $\lambda x : \mathsf{Int}.x \geq 9$, or $\lambda x : \mathsf{Int}.\mathsf{true}$.

Example 3.3. As an example, Unify is applied on the base case of the `compose` function from § 2.3.4, on the input $\mathsf{prop}\ p_2 = \texttt{Program}\ s_2 \wedge \mathsf{prop}\ p_2 = \texttt{Program id}$ and with a set of variables xs that includes s_2, but not the reflected `id`. The transitive closure of the candidates is the following four equalities: 1) $\mathsf{prop}\ p_2 \cong \mathsf{prop}\ p_2$, 2) $\mathsf{prop}\ p_2 \cong \texttt{Program}\ s_2$, 3) $\mathsf{prop}\ p_2 \cong \texttt{Program id}$, and 4) $\texttt{Program}\ s_2 \cong \texttt{Program id}$. The first three equalities will not result in any unifications since they contain function applications. The forth one instead is unifying the application of the `Program` constructor on both sides, hence the arguments s_2 and `id` can be unified. Since s_2 belongs in the set of variables xs while `id` does not, the algorithm returns the unification $s_2 \cong \texttt{id}$.

Soundness of the Unification Algorithm. We proved that our decidable unification algorithm both terminates and also it is a sound approximation of the complete unifier.

Theorem 3.4 (Unify). *For any* xs *and* r, $\mathsf{Unify}(xs, r) \subseteq \mathcal{U}(r)$ *and* $\mathsf{Unify}(xs, r)$ *terminates.*

3.4 Metatheory

Finally, we show that the typing rules are sound, *i.e.,* if a program type checks with a type τ, then the program is in the denotation of the type τ. Crucially, soundness only holds if the reflection and unification environments are well formed *w.r.t,* the typing environment. We call the unification environment well formed if it only contains equalities that respect the typing environment.

Definition 3.5 (Well Formedness of Unification Environment). The well formedness of the unification environment U, *w.r.t,* a typing environment Γ and a reflection environment R, is defined as follows:

$$\Gamma; R \models U \doteq \forall x \cong e_U \in U, \forall \theta \in [[\Gamma]].\ x \in \mathsf{dom}(\Gamma) \wedge x \notin \mathsf{dom}(R) \wedge \theta(x) = e_x \Rightarrow e_U = e_x.$$

That is, a unification environment is well formed if for every binding $x \cong e_U$ in the unification environment, the variable x is in the domain of the typing environment Γ, and every closing substitution θ of Γ, binds x to an expression e_x equal to e_U. Further, we require that all the free variables in the equalities are not bound in the reflection environment, since it already contains the definition of these functions. Well formedness of the reflection environment requires that all the reflected functions are well-typed with respect to the typing environment.

Definition 3.6 (Well Formedness of Reflection Environment). The well formedness of the reflection environment R, *w.r.t,* a typing environment Γ and a unification environment U, is defined as follows:

$$\Gamma; U \models R \doteq \forall x : \tau := e \in R, \Gamma, x : \tau; R; U \vdash e : \tau$$

$$
\begin{array}{rrl}
\textbf{Bin-Ops} & \oplus & ::= \wedge \mid + \mid - \mid = \mid < \mid \ldots \\
\textbf{Constants} & c & ::= \mathsf{true} \mid \mathsf{false} \mid 0, -1, 1, \ldots \\
\textbf{Terms} & t & ::= t \oplus t \mid \neg t \mid x \mid c \mid x\ \overline{t} \mid D\ \overline{t} \\
 & & \mid\ \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \lambda\!\!\lambda x\!:\!\tau.t \mid t\ t
\end{array}
$$

$$
\begin{array}{rrl}
\textbf{Environments} & & \\
\textbf{Logical} & \Phi & := \emptyset \mid \Phi, t, t \bowtie t \\
\textbf{Reflection} & \Psi & := \emptyset \mid \Psi, f\ \overline{y} := t \\
\textbf{Delta} & \Delta & := \emptyset \mid \Delta, f \cong t
\end{array}
$$

Fig. 6. Syntax of the Logical Calculus λ^S. To encode λ^S terms to SMT-LIB, $\lambda\!\!\lambda x\!:\!\tau.t$ is written as $\mathtt{lam}\ t$ and $t\ t$ as $(\mathsf{app}\ t\ t)$, where lam and app are a family of uninterpreted functions.

That is, all reflected definitions can be typed under the typing environment.

Finally, we prove the soundness of the system. Namely, that if a program type checks with a type τ, under well formed environments, then the program is in the denotation of the type.

Theorem 3.7 (Soundness). *For any* Γ, U, R, e, τ *such that* $\Gamma; U \models R$ *and* $\Gamma; R \models U$, *if* $\Gamma; R; U \vdash e : \tau$ *then* $\forall \theta \in [[\Gamma]]\ .\ \theta \cdot e \in [[\theta \cdot \tau]]$.

4 Proof by Logical Evaluation Extended with $\beta\eta\delta$ Equivalence

Proof by Logical Evaluation (PLE [Vazou et al. 2017]) is a technique that allows to symbolically (*i.e.,* using variables) and logically (*i.e.,* using an SMT solver) evaluate expressions. In this section we extend the PLE algorithm to work with β- and η-equivalences, and a new δ rule that applies the unifiers. Concretely, we define a symbolic core calculus λ^S (§ 4.1) and reduce λ^R into it, then we define the PLEX algorithm (§ 4.2), and we prove its soundness and termination (§ 4.3). Finally, we use PLEX to decide entailment (§ 4.4) and obtain a sound and decidable refinement type system.

4.1 Core Calculus λ^S

Figure 6 defines the syntax of λ^S, a logical calculus that is designed to approximate λ^R and to be easily embedded into a subset of the SMT-LIB's decidable fragment.

Operators of λ^S ($\oplus$) include the same operators as in λ^R, *i.e.,* for booleans and integer linear arithmetic. These operators belong to decidable fragments of SMT-LIB and are all binary operators, since λ^S does not permit partially applied operators.

Terms of λ^S (t) include fully applied binary operators ($t \oplus t$), logical negation ($\neg t$), variables (x), and constants (c) for integers and booleans. Terms also contain applications of data constructors $D\ \overline{t}$, uninterpreted functions $x\ \overline{t}$, and the if expression. All these elements are part of the decidable fragment of SMT-LIB. λ^S contains also lambda abstractions $\lambda\!\!\lambda x\!:\!\tau.t$ typed with a refinement type from λ^R. Given that λ^S is a logic language, these abstractions do not come with the standard reduction rules, that we approximate later, instead they are encoded in SMT-LIB using a family of the uninterpreted function lam, one for each possible SMT sort of the body t. Finally, λ^S terms contain function applications $t\ t$, that simulate the λ-application and are encoded in SMT-LIB using a family of uninterpreted functions app.

4.1.1 Embedding of λ^R into λ^S into SMT-LIB. We define the function $\lfloor e \rfloor$ that embeds an expression e of λ^R into a term t of λ^S. Below we present the key cases of the embedding function:

$$
\begin{array}{llll}
\lfloor c \rfloor \doteq c & \lfloor \lambda x\!:\!\tau.e \rfloor \doteq \lambda\!\!\lambda x\!:\!\tau.\lfloor e \rfloor & \lfloor D\ \overline{e} \rfloor \doteq D\ \overline{\lfloor e \rfloor} & \\
\lfloor x \rfloor \doteq x & \lfloor \oplus\ e_1 e_2 \rfloor \doteq \lfloor e_1 \rfloor \oplus \lfloor e_2 \rfloor & \lfloor \neg e \rfloor \doteq \neg \lfloor e \rfloor & \lfloor e_1\ e_2 \rfloor \doteq \lfloor e_1 \rfloor\ \lfloor e_2 \rfloor \\
\multicolumn{4}{l}{\lfloor \mathsf{case}\ x = e\ \mathsf{of}\{D_i\ \overline{x} \rightarrow e_i\} \rfloor \doteq \mathsf{if}\ \mathsf{isD}_1\ \lfloor e \rfloor\ \mathsf{then}\ \lfloor e_i \rfloor [\lfloor e \rfloor / x] [\mathsf{getD}_{1_j}\ \lfloor e \rfloor / \overline{x}]\ \mathsf{else}\ \ldots}
\end{array}
$$

Constants, variables, and primitives are embedded directly into λ^S. Undef will never be embedded to the SMT language. Data constructors are embedded as applications of the data constructor. Case expressions are embedded into if-then-else expressions using the uninterpreted function isD to

check that the scrutinee is a data constructor and getD to extract the arguments of the data constructor, while the embedding assumes that all branches of the case expression are present.

Lambda Abstractions. Since λ^S is a first-order, logical language, it does not have interpreted λ abstractions. To embed λ abstractions of λ^R we use the intermediate (between λ^R and SMT-LIB) representation $\lambda x{:}\tau.\lfloor e \rfloor$, that preserves the refinement types. To embed $\lambda x{:}\tau.t$ to the SMT-LIB we use a family of uninterpreted functions `lam`, one for each possible SMT sort of the argument x and body t, and embed $[\![\lambda x{:}\tau.t]\!] = \texttt{lam}\ [\![t[i_0^{[\![\tau]\!]}/x]]\!]$ where $[\![\tau]\!]$ is the SMT sort of τ and each i_j^s is a fresh constant for each sort s, and de Bruijn index j, ensuring α-equivalent terms embed identically.

Function Applications. Dually to the λ abstractions, function applications are also embedded into an intermediate application and then to the SMT-LIB using an uninterpreted function. Suppose for example that we need to embed the application $(\lambda x{:}\tau.x)\ 42$. Since $\lambda x{:}\tau.x$ is embedded as $\lambda x{:}\tau.x$, and encoded into SMT-LIB as $\texttt{lam}\ i_0^{[\![\tau]\!]}$, we cannot simply use the SMT-LIB application, instead we use the uninterpreted function `app`, that takes two arguments, the function and the argument. Thus, the application $(\lambda x{:}\tau.x)\ 42$ is embedded to SMT-LIB as $\texttt{app}\ (\texttt{lam}\ i_0^{[\![\tau]\!]})\ 42$. Our embedding, using uninterpreted functions for both λ-abstractions and applications, allows the encoding of all λ^R expressions, including higher-order and partially applied functions, into the logical language λ^S and then into SMT-LIB. Yet, our embedding carefully distinguishes between applications of function that are reduced to first-order *e.g.,* via the usage of `lam`, and the "real", interpreted function applications. When embedding applications of data constructors and interpreted symbols, like +, =, …, it preserves the direct application. Otherwise, it uses the uninterpreted function `app`. The detailed embedding, presented in the appendix, uses type directed rules to ensure that the final result is well sorted and accepted by an SMT solver. Here, for simplicity of exposition we collapse the two cases of application into one.

Embedding of Environments. A λ^R typing environment Γ contains bindings of variables to refined types. We embed a typing environment Γ into a set of λ^S terms, by only embedding the predicates of each base refined type, and ignoring the function types that do not have top level refinements. Similarly, we embed the reflection environment R and the unification environment U, by embedding the bodies of the definitions.

$$\begin{array}{lcl} \lfloor \Gamma \rfloor & \doteq & \{\lfloor r[x/v] \rfloor \mid (x, \{v{:}B \mid r\}) \in \Gamma\} \\ \lfloor U \rfloor & \doteq & \{x \cong \lfloor e \rfloor \mid x \cong e \in U\} \\ \lfloor R \rfloor & \doteq & \{f\ \overline{y} := \lfloor e \rfloor \mid f : (\overline{y:\tau} \to \tau) := e \in R\} \end{array}$$

Semantic Preservation. In Theorem 4.2 (I) we prove that our embedding definition preserves the semantics of the λ^R language. Intuitively that if a term e evaluates to a value w, then the embedding of e is semantically equal to the embedding of w. We proved that the two embeddings are equal, under the assumptions of the embedding of some environment Γ that contains the free variables of e, extended with axioms that capture the computational properties of the system, *i.e.,* β and η-equivalence.

Definition 4.1 (Computational Extension). Given a set of formulas Φ its computational extension Φ^+ is defined as $\Phi \cup \{\alpha, \beta, \eta\}$; where α: $\forall x, y, t, \tau.\ \lambda x{:}\tau.t = \lambda y{:}\tau.t[x/y]$, β: $\forall x, \tau, t, t_x.\ \lambda x{:}\tau.t\ t_x = t[x/t_x]$, and η: $\forall e$. if e has type $x{:}\tau_x \to \tau$ then $\lfloor e \rfloor = \lambda x{:}\tau_x.\lfloor e \rfloor\ x$.

We use the notation $\models_\Phi t$ to denote that the formulas in the set Φ logically imply the formula t. We show that (*I*) if an expression e evaluates to a value w under any closing substitution of Γ, then the embedding of Γ implies that the embedding of e is equal to the embedding of w. (*II*) If an expression is equal to some other value w', then w is equal to w' up to $\alpha\eta$-equivalence. Finally,

$$
\begin{array}{lcl}
\hline
\text{PLEX} & : & (\Gamma, \Phi, \Psi, \Delta, t) \to \text{Bool} \\
\hline
\text{PLEX}(\Gamma, \Phi, \Psi, \Delta, t) & \doteq & \text{loop}(0, \Phi \cup \{t_1 = t_2 \mid t_1 \in \text{C}_\Phi(t) \text{ and } \Gamma; \Phi; \Psi; \Delta \vdash t_1 \bowtie t_2\}) \\
\quad \texttt{where} \quad \text{loop}(i, \Phi_i) & \mid & \Phi_i \models_{\text{SMTValid}} t \doteq \texttt{true} \\
& \mid & \Phi_{i+1} \subseteq \Phi_i \doteq \texttt{false} \\
& \mid & \texttt{otherwise} \doteq \text{loop}(i+1, \Phi_{i+1}) \\
& & \quad \texttt{where} \quad \Phi_{i+1} \doteq \text{Unfold}(\Gamma, \Phi_i, \Psi, \Delta) \\
\hline
\text{Unfold} & : & (\Gamma, \Phi, \Psi, \Delta) \to \Phi \\
\hline
\text{Unfold}(\Gamma, \Phi, \Psi, \Delta) & \doteq & \Phi \cup \{t_1 = t_2 \mid t_1 \in \text{C}_\Phi(\Phi), \Gamma; \Phi; \Psi; \Delta \vdash t_1 \bowtie t_2\} \\
\hline
\end{array}
$$

Term Reduction $\boxed{\Gamma; \Phi; \Psi; \Delta \vdash t \bowtie t}$

$$\frac{\Phi \models_{\text{SMTValid}} t}{\Gamma; \Phi; \Psi; \Delta \vdash (\text{if } t \text{ then } t_1 \text{ else } t_2)\ \overline{t_a} \bowtie t_1\ \overline{t_a}}\ \text{I-TIF} \qquad \frac{f \cong t_f \in \Delta}{\Gamma; \Phi; \Psi; \Delta \vdash f\ \overline{ts} \bowtie t_f\ \overline{ts}}\ \text{I-DELTA}$$

$$\frac{\Phi \models_{\text{SMTValid}} \neg t}{\Gamma; \Phi; \Psi; \Delta \vdash (\text{if } t \text{ then } t_1 \text{ else } t_2)\ \overline{t_a} \bowtie t_2\ \overline{t_a}}\ \text{I-FIF} \qquad \frac{(f, \overline{y}, t) \in \Psi \quad \mid \overline{ts_1} \mid = \mid \overline{y} \mid}{\Gamma; \Phi; \Psi; \Delta \vdash f\ \overline{ts_1}\ \overline{ts_2} \bowtie t[\overline{ts_1}/\overline{y}]\ \overline{ts_2}}\ \text{I-REFL}$$

$$\frac{}{\Gamma; \Phi; \Psi; \Delta \vdash \lambda x{:}\tau.t\ t_x\ \overline{ts} \bowtie t[t_x/x]\ \overline{ts}}\ \text{I-BETA}$$

$$\frac{\Gamma(f) = \overline{x{:}\tau_x} \to \tau \quad m = \left|\overline{ts}\right| \quad n = |\overline{\tau_x}| \quad n > m \quad \texttt{fresh}\ y_i}{\Gamma; \Phi; \Psi; \Delta \vdash f\ \overline{ts} \bowtie \lambda y_{m+1}{:}\tau_{m+1}[\overline{y}/\overline{x}]. \ldots \lambda y_n{:}\tau_n[\overline{y}/\overline{x}].f\ \overline{ts}\ y_{m+1}\ \ldots\ y_n}\ \text{I-ETA}$$

Fig. 7. The PLEX algorithm.

(*III*) if an expression is a boolean term and its embedding is valid, then the expression evaluates to true under any closing substitution of Γ.

Theorem 4.2 (Semantic preservation). *The embedding establishes a connection between the logical models of λ^R and the operational semantics of λ^S:*

(I) *Value Preservation: If* $\forall \theta \in [[\Gamma]].\ \theta \cdot e \hookrightarrow^* w$ *then* $\models_{\lfloor \Gamma \rfloor^+} \lfloor e \rfloor = \lfloor w \rfloor$.
(II) *$\alpha\eta$ Preservation: If* $\forall \theta \in [[\Gamma]].\ \theta \cdot e \hookrightarrow^* w$, *then* $\models_{\lfloor \Gamma \rfloor^+} \lfloor e \rfloor = \lfloor w' \rfloor \Rightarrow w' =_{\alpha\eta} w$.
(III) *True Preservation: If* $\Gamma; R; U \vdash e : \text{Bool}^{\Downarrow}$ *and* $\models_{\lfloor \Gamma \rfloor^+} \lfloor e \rfloor$, *then* $\forall \theta \in [[\Gamma]].\ \theta \cdot e \hookrightarrow^* \texttt{true}$.

4.2 The PLEX Algorithm

Figure 7 presents the PLEX algorithm, a Proof by Logical Evaluation algorithm, extended with β and η-equivalence and unification. PLEX, essentially uses an SMT solver to evaluate boolean expressions, thus incorporating the computational reduction rules to the SMT solving.

The PLEX algorithm takes as input five arguments:

1) The typing environment Γ that contains the refinement types of the free variables.
2) The logical environment Φ that contains hypotheses, *i.e.,* a set of terms that are assumed to hold.
3) The reflection environment Ψ that maps function symbols to their definitions, *i.e.,* the set of triples $f\ \overline{y} := t$ where f is a function symbol, $\overline{y}$ are the arguments of f, and t is the body of the definition. The Ψ is the embedding of the λ^R reflection environment.

$$
\begin{array}{ll}
\mathsf{C}_\Phi(\oplus_1 t) \doteq \{\oplus_1 t\} \cup \mathsf{C}_\Phi(t) & \mathsf{C}_\Phi(t_1 \oplus t_2) \doteq \{t_1 \oplus t_2\} \cup \mathsf{C}_\Phi(t_1) \cup \mathsf{C}_\Phi(t_2) \\
\mathsf{C}_\Phi(b) \doteq \emptyset & \mathsf{C}_\Phi(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) \doteq \{\text{if } t_1 \text{ then } t_2 \text{ else } t_3\} \cup \mathsf{C}_\Phi(t_1) \\
\mathsf{C}_\Phi(n) \doteq \emptyset & \mathsf{C}_\Phi(t_1\ \overline{t}_2) \doteq \{t_1\ \overline{t}_2\} \cup \bigcup_i \mathsf{C}_\Phi(t_{2_i}) \\
\mathsf{C}_\Phi(x\ \overline{t}) \doteq \{x\ \overline{t}\} \cup \bigcup_i \mathsf{C}_\Phi(t_i) & \mathsf{C}_\Phi(\lambda x{:}\tau.t) \doteq \begin{cases} \mathsf{C}_{\Phi \cup Guard(\tau)}(t) & if\, \Phi \models_{\mathsf{SMTSat}} Guard(\tau) \\ \emptyset & otherwise \end{cases} \\
\mathsf{Guard}(\{x : B \mid r\}) \doteq \lfloor r \rfloor & \mathsf{Guard}(x{:}\tau_x \to \tau) \doteq \mathsf{Guard}(\tau)\ if\ x \notin \mathtt{fv}(\mathsf{Guard}(\tau))\ else\ false
\end{array}
$$

Fig. 8. Application Candidates.

4) The delta environment Δ that maps symbols to their definitions, *i.e.,* the set of pairs $f \cong t_f$ where f is equivalent to t_f. It is called delta environment, since it is used to bind equivalences derived from the pattern matching, thus mimicking dependent pattern matching.
5) The final input is a λ^S term t, whose validity is to be decided by the algorithm.

PLEX$(\Gamma, \Phi, \Psi, \Delta, t)$ returns `true` if the term t is valid in the logical environment Φ and the reduction rules. To decide that, reduction rules are encoded as equalities which extend the logical environment. Of course, reductions can lead to further reductions, thus PLEX is designed as an iterative algorithm that alternates between applying the reductions and checking the SMT validity, until the term is valid or no new reductions can be applied.

The algorithm is using two main procedures, the term reduction $\Gamma; \Phi; \Psi; \Delta \vdash t_1 \bowtie t_2$ that derives the term t_2 to which t_1 can be reduced and the application candidates $\mathsf{C}_\Phi(\cdot)$ that returns the set of terms that are reducible, *i.e.,* some reduction can be applied to them.

The Term Reduction relation $\Gamma; \Phi; \Psi; \Delta \vdash t_1 \bowtie t_2$, defined in fig. 7, reduces the term t_1 to t_2. The rule I-FIf reduces the `if` expression to the then branch if the condition is decided valid by the SMT solver, under the logical environment Φ. Dually, the rule I-FIf reduces the `if` expression to the else branch if the negated condition is decided valid. The rule I-Beta reduces a function application to the body of the function, using the standard β-reduction. The rule I-Refl reduces a function application to the body of the function, using the reflection environment Ψ. The rule I-Delta reduces a function application to the body of the function, using the equivalences in the delta environment Δ. Finally, the rule I-Eta performs η-expansion on a function application, by generating fresh variables for the missing arguments; using the typing environment Γ to obtain the types of the arguments since λ abstractions are typed.

The Application Candidates function $\mathsf{C}_\Phi(\cdot)$, defined in fig. 8, returns the longest application chains in a term. Most cases are trivial and simply return the term itself and the candidates of its subterms; the non-trivial cases are three: if-then-else, λ abstractions, and function applications. 1) In the if-then-else terms the candidates are the whole if-then-else expression and the guard, since unfolding the branch without assuring the condition could lead to divergence, in the case of a recursive function. 2) When selecting candidates for a λ abstraction, the body is selected only if the refinement of the λ argument is *satisfiable*; otherwise the body cannot be reached. Satisfiability is checked by the $\mathsf{Guard}(\tau)$ predicate that if τ is a base type, it returns its refinement, otherwise, it over-approximates the refinement of the functional type to the refinement of the return type. 3) For function applications only the complete application is selected as a candidate. For example, in the term `f x y`, we select only `f x y` and not also `f x`. This selection ensures that partial function applications are transformed in η-long form. We abuse the notation $\mathsf{C}_\Phi(\Phi)$ to denote $\bigcup_{t \in \Phi} \mathsf{C}_\Phi(t)$.

The PLEX Algorithm, defined in fig. 7, is using a $\mathsf{loop}(i, \Phi_i)$ function that at each iteration i, checks that the term t is valid in the logical environment Φ, in which case it returns `true`. Otherwise, it computes the next logical environment Φ_{i+1}, by adding the equalities obtained by reducing the

$$\frac{\text{PLEX}(\Gamma, \lfloor \Gamma, x : \{x{:}B \mid e_1\} \rfloor, \lfloor R \rfloor, \lfloor U \rfloor, \lfloor e_2 \rfloor)}{\Gamma; R; U \vdash \{x{:}B \mid e_1\} \Rightarrow \{x{:}B \mid e_2\}} \text{ I-PLEX}$$

Fig. 9. Decidable Entailment Checking with PLEX.

application candidates of the environment Φ_i. If no new equalities are obtained, the algorithm returns `false`, otherwise, it continues to the next iteration.

4.3 Soundness and Termination of PLEX

Here, we prove that PLEX is sound, with respect to operational semantics of λ^R, and terminating.

Soundness states that if the PLEX returns `true` for an embedding of a type environment Γ and a term e, then the term e reduces to true under any closing substitution of Γ:

Theorem 4.3 (Soundness). *Soundness is stated on three levels logical, term reduction, and PLEX:*

(I) *Logical Model: If* $\Gamma; U \models R, \Gamma; R \models U$ *and PLEX*$(\Gamma, \lfloor \Gamma \rfloor, \lfloor R \rfloor, \lfloor U \rfloor, \lfloor e \rfloor)$*, then* $\models_{\lfloor \Gamma \rfloor^+} \lfloor e \rfloor$.

(II) *Term Reduction: If* $\models_{\lfloor \Gamma \rfloor^+} \Phi, \Gamma; U \models R, \Gamma; R \models U$*, and* $\Gamma; \Phi; \lfloor R \rfloor; \lfloor U \rfloor \vdash t_1 \bowtie t_2$*, then* $\models_{\lfloor \Gamma \rfloor^+} t_1 = t_2$.

(III) *PLEX: If* $\Gamma; U \models R, \Gamma; R \models U$ *and PLEX*$(\Gamma, \lfloor \Gamma \rfloor, \lfloor R \rfloor, \lfloor U \rfloor, \lfloor e \rfloor)$*, then* $\forall \theta \in [\![\Gamma]\!].\ \theta \cdot e \hookrightarrow^* \text{true}$.

To prove the soundness of PLEX (*i.e.,* Theorem 4.3 (III)) we first proved that the term reduction relation is sound with respect to the logical model. By consequence of Theorem 4.2 (III) we have that PLEX is sound with respect to the operational semantics.

The proof of termination works by associating the equalities obtained by PLEX to reduction steps of the original program. In particular we show that any sufficiently long chain of equalities implies an evaluation step of one of the terms that are in $\lfloor \Gamma \rfloor$, hence having an infinite amount of equalities is equivalent to having an infinite amount of reduction steps contradicting the assumption that all the terms that are in refinements are terminating.

Theorem 4.4 (Termination). *If* $\lfloor \Gamma \rfloor \models_{\text{SMTValid}} \mathit{False}$ *or* $|[\![\Gamma]\!]| \geq 1$ *(i.e.,* Γ *is* transparent *[Vazou et al. 2017]),* $\Gamma; U \models R, \Gamma; R \models U, \Gamma; R; U \vdash e : \tau$*, then PLEX*$(\Gamma, \lfloor \Gamma \rfloor, \lfloor R \rfloor, \lfloor U \rfloor, \lfloor e \rfloor)$ *is terminating.*

4.4 Entailment Checking with PLEX

We use PLEX to check entailment of two refined types, thus modifying the typechecking rules of fig. 3 to be decidable. Figure 9 presents the rule I-PLEX that uses the PLEX algorithm to check entailment. It converts the environments to their λ^S embeddings, using $\lfloor \cdot \rfloor$, and checks that they are sufficient to show the validity of e_2. We proved that the derived refinement type system is both sound, as a corollary of soundness of both PLEX (Theorem 4.3 (III)) and Unify (Theorem 3.4), and terminating, as a corollary of the termination of PLEX and Unify (Theorems 4.4 and 3.4).

5 Implementation & Case Studies

We have implemented the PLEX algorithm as an extension to Liquid Haskell [Vazou et al. 2014]. The implementation (§ 5.1) consists of incorporating the unifier (of § 3.3) in the refinement type checker and the reduction rules (of § 4.2) in the PLE algorithm. We developed eight case studies (§ 5.2) and we used them to compare PLEX to the original PLE (§ 5.3) as well as `Agda` (§ 5.4). Since PLEX extends refinement type checking with two features: 1) normalization (β and η rules) and 2) dependent pattern matching, while preserving all the existing features (*e.g.,* interaction with the SMT), our benchmarks evaluate only the two additional features and make very little use of the SMT support. We chose to compare against `Agda`, because it is a widely used dependently typed programming language with well-documented support for automated dependent pattern

Table 1. PLEX case studies: δ and $\beta\eta$ indicate whether the case study required, respectively, δ-reduction and $\beta\eta$-equivalences. **Exec**, **Spec**, and **Proofs** give lines of code for the executable, the specification, and the explicit proofs. **Agda LoC** and **Agda Time (s)** report the size and type checking time of the Agda developments.

	PLEX						Agda	
Benchmark	δ	$\beta\eta$	**Exec**	**Spec**	**Proof**	**Time (s)**	**LoC**	**Time (s)**
`Stack Compiler` (§ 2)	✓	✓	53	19	0	3.558	69	6.727
`STLC`	✓	✓	95	70	0	4.433	70	2.335
`Compiler Exceptions`	×	✓	85	58	0	3.963	84	3.193
`Map Fusion` (Rewrite Rules)	×	✓	22	12	6	1.904	27	1.173
`List Monad`	×	✓	28	16	11	2.235	59	1.351
`Reader Monad`	×	✓	13	9	4	2.110	29	1.084
`State Monad`	×	✓	12	8	3	2.151	31	1.214
`Continuation Monad`	×	✓	11	8	3	2.131	25	1.092
Total			319	200	27	**22.584**	**394**	**17.197**

matching [Cockx and Abel 2018]. Finally, in § 5.5 we present the limitations of PLEX, and how the user can recover in case of failure.

5.1 Implementation

`Liquid Haskell`'s refinement type checking relies on two calculi. The first is `CoreSyn` [Sulzmann et al. 2007], `GHC`'s intermediate language, which is a superset of λ^R and is used by `Liquid Haskell` to perform refinement type checking based on rules similar to those presented in § 3.2. A rule, similar to $\preceq$-Base in fig. 3, reduces refinement type checking to implication checking. These implications are then transformed into terms of the liquid-fixpoint logical calculus, similar to λ^S, which serves as the intermediate layer between `CoreSyn` and the `SMT-Lib` calculus.

The refinement type checking is performed in various stages, including:

- **Refinement Type Checking** that reduces type checking of λ^R to implication checking of λ^S.
- **Proof By Logical Evaluation** that reduces evaluation of the reflected functions.
- **Refinement Inference** attempts to infer the unknown refinements [Rondon et al. 2008].
- **Validity Checking** is using the SMT solver to decide the validity of the logical formulas.

Our implementation only modified the first two stages of the `Liquid Haskell` pipeline, which indicates that the extension is relatively modular and can be integrated to more SMT-based verification tools. We modified the refinement type checking stage to keep track of the unification environment and perform the unification procedure from § 3.3 when checking case expressions, which required about 138 new lines of code. Additionally, we extended the implementation of PLE inside `Liquid Haskell` to incorporate the reduction rules of PLEX in 698 lines of new code. Finally, we extended the existing calculus to support λ abstractions in the refinements.

5.2 Case Studies

Table 1 summarizes the case studies used to evaluate PLEX. For each study, we report six columns: δ and $\beta\eta$ indicate whether dependent pattern matching and $\beta\eta$-equivalence were required. The other three columns show Lines of Code (LoC) for the executable (Haskell functions), the specification (refinement type annotations), and any explicit proofs (bodies of the specifications). Finally, `time` reports the verification time. We used eight case studies. The first is the correct by construction compiler of § 2 and the second translates the simply typed lambda calculus to combinatory

logic [Swierstra 2023]. As illustrated in § 2, these case studies use data propositions to encode the semantics of the programs and the dependent pattern matching, *i.e.,* the δ rule, is required to avoid hand-written proofs. The third translates the compiler with exceptions from [Pickard and Hutton 2021]. It uses indexed families to safely compose stack machine programs while ensuring proper exception handling. The fourth case study establishes the correctness of the GHC rewrite rules for map fusion. Concretely, GHC rewrites saturated map applications with a `build/fold` form hoping for fusion to happen, applying the following four rewrite rules, that we proved correct:

```
{-# RULES "map"       ∀ f xs.  map f xs = build (\c n → foldr (mapFB c f) n xs)
          "mapList"   ∀ f.     foldr (mapFB (:) f) []  = map f
          "mapFB"     ∀ c f g. mapFB (mapFB c f) g     = mapFB c (f.g)
          "mapFB/id"  ∀ c.     mapFB c (\x → x)        = c                  #-}
```

The last four case studies prove the three monad laws, *i.e.,* left identity, right identity, and associativity, for the list, reader, state, and continuation monads, including all the lemmata required to conduct these proofs, like associativity of list append and that bind distributes over append.

5.3 Comparison with the PLE Algorithm

The extensions we implemented to PLE are backwards compatible, *i.e.,* all the previous proofs remain valid, and, impose a small slowdown on verification. Concretely, we run PLEX on the existing benchmark suite of `Liquid Haskell` and observed a 5% slowdown. Moreover, none of the proofs of Table 1 are feasible without our extension. In fact, with the vanilla PLE, the monad laws can only be proved by admitting the function extensionality axiom. This axiom was later found to be inconsistent with the refinement type logic [Vazou and Greenberg 2022]. Although Vazou and Greenberg provide a type-based, and thus consistent, extensionality axiom, their work also demonstrates that developing proofs with this axiom is cumbersome; furthermore, it is not compatible with the PLE automation. That is, the type-based equalities derived by the extensionality axiom are not automatically accepted by PLE. On the contrary, PLEX incorporates the β and η equalities at the level of λ^S, where the refinement types are not present. As a result, it is not affected by the inconsistency between the refinement type logic and extensionality axiom.

5.4 Comparison with Agda

To evaluate the effectiveness of PLEX, we ported the case studies to Agda, and in Table 1 we report the length and verification time of the proofs. In both cases, the lengths of the proofs are similar, especially taking into account that the `Liquid Haskell` developments require extra effort since the final developments are also valid Haskell programs, so declarations need to be duplicated and verification time also counts the `ghc` compilation time. The essential differences are that in PLEX the proofs use subtyping (§ 5.4.1) and SMT automation (§ 5.4.3), while the Agda proofs use implicit arguments (§ 5.4.2) and large elimination (§ 5.4.4). Next, we elaborate on these differences.

5.4.1 Implicit Subtyping. Agda's type system does not allow implicit subtyping. As a result, many proofs require extra definitions that can be implicit in `Liquid Haskell`. For example, in the `Stack Compiler` case study the Agda version requires explicit tracking of stack size, adding two additional indices to the `Program` data type: one for the stack size before execution and one for the stack size after execution. In `Liquid Haskell` the stack size is automatically computed using refinements on the length of the stack, without the need for additional indices.

5.4.2 Implicit Arguments. Conversely, Agda allows implicit arguments to be automatically inferred. This helps keep the complexity of large type signatures in check. In `Liquid Haskell`, implicit arguments are very challenging to implement [Tondwalkar et al. 2021]. The main issue is that, by design, Liquid Haskell operates as a layer over Haskell, where the source code must remain

valid Haskell. As Haskell itself does not have a similar feature, there is no easy way to add such implicit arguments to Liquid Haskell's types. The STLC example also makes heavy use of implicit arguments, which is the main reason why the Agda version is shorter than the Liquid Haskell.

5.4.3 SMT Automation. Agda does not have the level of proof automation that Liquid Haskell does. As Liquid Haskell translates its verification conditions to SMT queries, many numeric proofs are discharged automatically. For example, in the compile function of § 2.4, the correctness of the negation operation is automatically proven in Liquid Haskell but must be explicitly proven in Agda. The Agda signature is presented below:

```
compile : ∀ {n} → (e : Expr) → Program n (1 N.+ n) (push $ eval e)
```

Proofs in Agda rely on *how* a function is defined: we cannot freely change `1 + n` to `n + 1` in the type of the compile function. Addition is defined by induction on its first argument; Agda does not automatically see that the definition is commutative. This is much less of a problem for Liquid Haskell, where such proof obligations are discharged automatically by the SMT solver.

5.4.4 Large Elimination. A notable difference between Agda and Liquid Haskell is the presence of large elimination. Since Liquid Haskell is a layer on top of Haskell it only accepts well typed Haskell programs. Instead one can use data propositions with negative occurrences but that are still well-founded. A concrete example of this phenomenon appears in the STLC case study, and concretely in the definition of the value type. In Agda, values are defined using large elimination:

```
data U : Set where          Val : U → Set
  ι : U                     Val ι = A
  _⇒_ : U → U → U           Val (u₁ ⇒ u₂) = Val u₁ → Val u₂
```

In Haskell directly porting such a definition in the expression level is not possible, since → cannot appear in the expression. Instead, we can use a data proposition to encode the same information, but we had to relax the positivity checker to allow the definition of Value. Despite this difference, we successfully mirrored the correct-by-construction approach in Liquid Haskell without additional lemmas or proofs beyond those defined in the compiler.

5.5 Limitations & Error Handling

PLEX is not complete, as it relies on higher-order unification that is known to be incomplete [Miller 1991]. Thus, the Unify procedure, from § 3.3, can also fail to find a unifier even when one exists. In such cases and since the user cannot interact with the unification to fix the error, the vanilla verification continues and returns a refinement type error, that the user must repair manually.

As an example where unification fails, we define the syntacticOr function that returns True if either of its arguments is True and construct a unification problem that requires to infer that `syntacticOr b c` ≅ False implies that b ≅ False. Concretely, consider the following code:

```
syntacticOr :: Bool → Bool → Bool {-@ reflect syntacticOr @-}
syntacticOr x y = case (x, y) of { (False, False) → False; _ → True }

{-@ data P where Slime :: b:Bool → c:Bool → Prop (syntacticOr b c) @-}

{-@ fail, pass :: Prop False → { v:Bool | not v } @-}
fail, pass :: P → Bool
fail (Slime b c) = b -- Unification Fails and defaults to vanilla type checking
pass (Slime b c) = case (b,c) of (False,False) → b -- SOLUTION: observe b and c
```

The code first defines and reflects the syntacticOr function. Then, it defines a data proposition with a single constructor Slime that holds when `syntacticOr b c` is True. The name refers to the

"green slime" phenomenon [McBride 2012] where indexing a type by a function leads to difficult unification problems.

Next, the functions `fail` and `pass` map a `P` value to a boolean. The refinement annotation indicates that the argument holds the property `False` and the boolean result must be `False`. The function `fail` relies only on PLEX's automation and fails to verify, since the unification fails to deduce that `b` is `False` from the fact that `syntacticOr b c` is `False`. The unification procedure does not interact with the user. Instead it will generate no unifiers and default to vanilla type checking, that will create a refinement type error. A similar example will also be rejected by `Agda`. The pattern match will be rejected since the required equality cannot be resolved definitionally. In Liquid Haskell, on the other hand, the pattern match is accepted, and to make the verification succeed the user may either explicitly pattern match on `b` and `c`, as in `pass`, or replace `syntacticOr` with a definition based on the built-in logical disjunction.

6 Related Work

Dependent Types. Programming languages based on dependent type theories, such as `Agda` [The Agda Development Team 2024] and `Rocq` [The Rocq Development Team 2024], all have some form of *conversion rule*, stating that two types that reduce to the same value are interchangeable. Similarly, PLEX defines a normalization-like procedure for refinement types. However, refinement type systems present unique challenges, not typically found in dependently typed languages. In particular, in dependently typed languages, structural termination guarantees strong normalization of terms. This makes it possible, for instance, to evaluate the individual branches of a pattern match or body of a λ abstraction, without the risk of making the normalization procedure non-terminating. In contrast, in refinement type systems like `Liquid Haskell` this is not guaranteed. These fundamental differences prevent this kind of unrestricted evaluation in refinement type systems, requiring a more nuanced approach to ensure termination of the normalization procedure.

Dependent Pattern Matching. Dependent pattern matching is different from pattern matching on simply typed values. Each branch of the case expression may uncover new information about the *value* of arguments before it, on which the type of the scrutinee depends. Dependent pattern matching, is a case expression that lets one use the information obtained by matching on the index of the type (together with the information of the data constructor which is permitted by just pattern matching). To make use of the information gained through dependent pattern matching, proof assistants typically elaborate pattern matching on a term as the application of the eliminator synthesized from the type definition [Goguen et al. 2006; Thierry 1992]. Inferring the motive of the eliminator can be implemented directly, as in `Agda` [Cockx and Abel 2018], or offered as an extension, as in `Rocq`'s `Equations` framework [Sozeau and Mangin 2019]. Even though `Equations` serve the same purpose as PLEX's δ rule, inferring the eliminator motive cannot be directly adopted by `Liquid Haskell`, as Haskell's core theory is not based on introduction and elimination rules. In addition the motive of the eliminator is synthesized from the type being constructed [McBride 2002]. This works because dependent types generally have a regular structure. Refinement types, on the other hand, do not follow this pattern: refinements can include arbitrarily complex propositions. Therefore, PLEX unification procedure explores the pattern matching information and ultimately hands the equalities to the SMT solver. `F*` has native support for dependent pattern matching, but fails to work with higher order indices. Thus, our first two examples that require such unification will fail. Finally, `Lean` has clean support for dependent pattern matching. It behaves similar to `Agda`, once the proper `simpl`/`unfold` annotations are added. So, `Lean`, like `Agda`, can verify all our table 1 benchmarks, with the proper annotations. Prior to PLEX, PLE [Vazou et al. 2017] was unable to use higher-order information from pattern matches and therefore failed to verify the benchmarks.

Dependent Types & SMT Automation. `Lean` and `Rocq` have respectively the tactics `lean-smt` [Mohamed et al. 2025] and `SMT-Coq` [Ekici et al. 2017] that use SMT solvers to discharge goals, but they differ fundamentally from PLEX. The PLEX algorithm interleaves the SMT solver and normalization techniques to solve the verification condition. On the contrary, both `SMT-Coq` and `lean-smt` tactics fire and forget: they translate the goal to SMT-LIB and pass it to the solver, who either can solve the goal completely or makes zero progress. But, there is no interaction between the SMT and the type checker from a proof automation perspective. There are two ways that `Lean` (or `Rocq`) with smt-tactics can fail compared to PLEX. First, the goal is higher order it wont be translated to SMT-LIB resulting in an error. Second, solving the goal, *e.g.*, one that wraps arithmetic reasoning inside lambdas, requires interaction between the SMT and the normalization techniques: neither normalization nor an smt tacit alone can solve such goals. Instead, PLEX, which is designed to interleave SMT with higher order reasoning, solves them automatically.

Type Based Verification. Refinement type systems adopt different approaches to verification. Flux [Lehmann et al. 2023] generates verification conditions solved by SMT solvers, while others extend typing rules with specific reasoning, such as lattice-based subtyping [Freeman and Pfenning 1991], equality [Sjöberg and Weirich 2015], or evaluation [Hamza et al. 2019]. `F*` [Swamy et al. 2016], like `Liquid Haskell`, uses inductive rules to generate verification conditions and defunctionalization to encode higher-order functions in first-order logic. Unlike `Liquid Haskell`, which is limited by Haskell, `F*` extends its base language with more expressive dependent types and provides tactics for direct evaluation of pure `F*` terms, with a similar goal to PLEX.

Higher-Order Logic inside SMT Solvers. SMT solvers provide limited support for higher-order logic. For example, Z3 encodes lambdas (with support for β and η equivalences) as arrays, yet, with "limited true higher order reasoning" [Z3 Team 2025]. As an example, Z3 returns `unknown` when asked to prove the validity of the formula `let f = \x y -> \z -> z in f 1 2 == id`. Failure on this simple example indicates that the Z3-lambda encoding is not a good selection for our benchmarks that heavily rely on higher-order functions. Further problems are that such an encoding is using quantifiers, making verification unpredictable, and precludes the interaction with our δ rule, making dependent pattern matching impossible. An alternative design could be to directly improve the higher-order encodings of SMT solvers. For example, using higher-order rewriting systems that have been extensively studied [Nipkow and Prehofer 1998] and some can even guarantee termination under confluence assumptions. The main challenge in such an approach would be that the encoding of functions does not preserve the type information. Instead, our approach uses the type information to select which candidates can be rewritten (fig. 8) preserving termination even in the presence of non-structurally terminating definitions. PLEX could be integrated as a theory within SMT solvers using techniques like Nelson and Oppen [1979]'s theory combination. However, this would require both careful engineering work and theoretical investigation to determine which theories satisfy the necessary conditions to be combined with PLEX. Therefore, such an integration is outside of the scope of this paper and is left as an interesting direction for future work.

7 Conclusion

We presented PLEX, an algorithm that extends refinement type checking to collect unification information from dependent pattern matching and then feeds them into an SMT-based, logical evaluation algorithm that uses them to reason about user-defined, terminating functions along with the η and β equivalence rules. We proved that PLEX is sound and terminating, implemented it in Liquid Haskell and evaluated it on eight higher-order examples. Our examples show that this extension facilitates higher-order reasoning in Liquid Haskell, and we hope that the development of PLEX inspires further research in the area of SMT-based verification of higher-order functions.

Acknowledgments

We thank Facundo Domínguez for the helpful discussions and code reviews and the anonymous reviewers for their insightful comments and suggestions. This work was funded by the Excellence Seal María de Maeztu Grant (CEX2024-001471-M funded by MICIU/AEI/10.13039/501100011033) and the ERC Grant CRETE (GA 101039196). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council; neither of which can be held responsible for them.

Data Availability

The artifact is publicly available at [Ferrarini et al. 2026]. It contains the appendix and references to the extended implementation of Liquid Haskell, along with benchmarks and formalizations used to validate the results. It includes verification examples in Liquid Haskell and Agda, as well as SMT-based comparisons using Lean-SMT and SMT-LIB.

References

P. Bahr and G. Hutton. 2015. Calculating Correct Compilers. *Journal of Functional Programming* (2015). https://doi.org/10.1017/S0956796815000180

G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. 2004. Type-based Termination of Recursive Definitions. *Mathematical Structures in Computer Science* (2004). https://doi.org/10.1017/S0960129503004122

J. F. Belo, M. Greenberg, A. Igarashi, and B. C. Pierce. 2011. Polymorphic Contracts. *ESOP* (2011). https://doi.org/10.1007/978-3-642-19718-5_2

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. 2011. Refinement Types for Secure Implementations. *TOPLAS* (2011). https://doi.org/10.1145/1890028.1890031

M. H. Borkowski, N. Vazou, and R. Jhala. 2024. Mechanizing Refinement Types. *POPL* (2024). https://doi.org/10.1145/3632912

J. Cockx and A. Abel. 2018. Elaborating Dependent (Co)pattern Matching. *ICFP* (2018). https://doi.org/10.1145/3236770

B. Ekici, A. Mebsout, C. Tinelli, C. Keller, G. Katz, A. Reynolds, and C. Barrett. 2017. SMTCoq: A plug-in for integrating SMT solvers into Coq. *CAV* (2017). https://doi.org/10.1007/978-3-319-63390-9_7

A. Ferrarini, N. Vazou, and W. Swierstra. 2026. Artifact for "PLEX: Normalization for Refinement Types". https://doi.org/10.5281/zenodo.19126672

T. Freeman and F. Pfenning. 1991. Refinement types for ML. *PLDI* (1991). https://doi.org/10.1145/113446.113468

C. Gamboa, P. Canelas, C. Timperley, and A. Fonseca. 2023. Usability-Oriented Design of Liquid Types for Java. *ICSE* (2023). https://doi.org/10.1109/ICSE48619.2023.00132

H. Goguen, C. McBride, and J. McKinna. 2006. Eliminating Dependent Pattern Matching. *Lecture Notes in Computer Science* (2006). https://doi.org/10.1007/11780274_27

J. Hamza, N. Voirol, and V. Kunčak. 2019. System FR: formalized foundations for the stainless verifier. *OOPSLA* (2019). https://doi.org/10.1145/3360592

R. Jhala and N. Vazou. 2021. Refinement Types: A Tutorial. *Found. Trends Program. Lang.* (2021). https://doi.org/10.1561/2500000032

N. D. Jones and N. Bohr. 2004. Termination Analysis of the Untyped λ-Calculus. *Rewriting Techniques and Applications* (2004). https://doi.org/10.1007/978-3-540-25979-4_1

K. Knowles and C. Flanagan. 2010. Hybrid Type Checking. *TOPLAS* (2010). https://doi.org/10.1145/1667048.1667051

N. Lehmann, A. T. Geller, N. Vazou, and R. Jhala. 2023. Flux: Liquid Types for Rust. *PLDI* (2023). https://doi.org/10.1145/3591283

K.M.R. Leino. 2010. Dafny: an automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*. https://doi.org/10.1007/978-3-642-17511-4_20

K.M.R. Leino and C. Pit-Claudel. 2016. Trigger Selection Strategies to Stabilize Program Verifiers. In *CAV*. https://doi.org/10.1007/978-3-319-41528-4_20

C. McBride. 2002. Elimination with a Motive. In *Types for Proofs and Programs*. https://doi.org/10.1007/3-540-45842-5_13

C. McBride. 2012. A Polynomial Testing Principle. PDF. https://personal.cis.strath.ac.uk/conor.mcbride/PolyTest.pdf

A. D. Miller. 1991. Unification of Simply Typed Lambda-Terms as Logic Programming. In *International Logic Programming Conference*. https://doi.org/20.500.14332/7376

A. Mohamed, T. Mascarenhas, H. Khan, H. Barbosa, A. Reynolds, Y. Qian, C. Tinelli, and C. Barrett. 2025. lean-smt: An SMT Tactic for Discharging Proof Goals in Lean. *CAV* (2025). https://doi.org/10.1007/978-3-031-98682-6_11

G. Nelson and D. C. Oppen. 1979. Simplification by Cooperating Decision Procedures. (1979). https://doi.org/10.1145/357073.357079

T. Nipkow and C. Prehofer. 1998. Higher-Order Rewriting and Equational Reasoning. In *Automated Deduction — A Basis for Applications. Volume I: Foundations.* https://doi.org/10.1007/3-540-48685-2_17

M. Pickard and G. Hutton. 2021. Calculating Dependently-typed Compilers (functional pearl). *ICFP* (2021). https://doi.org/10.1145/3473587

J. C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*. https://doi.org/10.1145/800194.805852

P. M. Rondon, M. Kawaguci, and R. Jhala. 2008. Liquid types. *PLDI* (2008). https://doi.org/10.1145/1379022.1375602

D. Sereni and N. D. Jones. 2005. Termination Analysis of Higher-Order Functional Programs. *APLAS* (2005). https://doi.org/10.1007/11575467_19

J. G. Siek, P. Thiemann, and P. Wadler. 2021. Blame and coercion: Together again for the first time. *Journal of Functional Programming* (2021). https://doi.org/10.1017/S0956796821000101

V. Sjöberg and S. Weirich. 2015. Programming up to Congruence. *POPL* (2015). https://doi.org/10.1145/2775051.2676974

M. Sozeau and C. Mangin. 2019. Equations Reloaded: High-level Dependently-typed Functional Programming and Proving in Coq. *ICFP* (2019). https://doi.org/10.1145/3341690

M. Sulzmann, M. Chakravarty, S. Peyton Jones, and K. Donnelly. 2007. System F with Type Equality Coercions. *PLDI* (2007). https://doi.org/10.1145/1190315.1190324

N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. Zinzindohoue, and S. Zanella-Béguelin. 2016. Dependent Types and Multi-monadic Effects in F*. *POPL* (2016). https://doi.org/10.1145/2837614.2837655

W. Swierstra. 2023. A correct-by-construction conversion from lambda calculus to combinatory logic. *Journal of Functional Programming* (2023). https://doi.org/10.1017/S0956796823000084

The Agda Development Team. 2024. *Agda wiki.* https://wiki.portal.chalmers.se/agda/Main/HomePage

The Rocq Development Team. 2024. *The Rocq Reference Manual – Release 8.19.0.* https://coq.inria.fr/doc/V8.19.0/refman

C. Thierry. 1992. Pattern Matching with Dependent Types. (1992). https://wonks.github.io/type-theory-reading-group/papers/proc92-coquand.pdf

A. Tondwalkar, M. Kolosick, and R. Jhala. 2021. Refinements of Futures Past: Higher-Order Specification with Implicit Refinement Types. *ECOOP* (2021). https://doi.org/10.4230/LIPIcs.ECOOP.2021.18

N. Vazou and M. Greenberg. 2022. How to Safely use Extensionality in Liquid Haskell. *Haskell Symposium* (2022). https://doi.org/10.1145/3546189.3549919

N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. 2014. Refinement Types for Haskell. *ICFP* (2014). https://doi.org/10.1145/2692915.2628161

N. Vazou, A. Tondwalkar, V. Choudhury, R. G. Scott, R. R. Newton, P. Wadler, and R. Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *POPL* (2017). https://doi.org/10.1145/3158141

H. Xi and F. Pfenning. 1998. Eliminating Array Bound Checking Through Dependent Types. *PLDI* (1998). https://doi.org/10.1145/277652.277732

Z3 Team. 2025. Z3 Guide - Lambdas. https://microsoft.github.io/z3guide/docs/logic/Lambdas/. Accessed: 2025-06-20.

Received 2025-10-09; accepted 2026-02-17