

PLEX: Normalization for Refinement Types

Making higher-order verification fit first-order SMT automation

Alessio Ferrarini^{1,2} Niki Vazou¹ Wouter Swierstra³

¹IMDEA Software Institute, Madrid, Spain

²Universidad Politécnica de Madrid, Madrid, Spain

³Utrecht University, Utrecht, Netherlands

OOPSLA 2026

We extend **Liquid Haskell** with PLEX,
a normalization procedure that makes proofs about
higher-order functions go through automatically,
while the SMT solver only sees **quantifier-free** queries.

Refinement types, automated by SMT

$\{-@ \text{div} :: \text{Int} \rightarrow \{y:\text{Int} \mid y \neq 0\} \rightarrow \text{Int} \ @-\}$

Refinement types, automated by SMT

$$\{-@ \text{div} :: \text{Int} \rightarrow \{y:\text{Int} \mid y \neq 0\} \rightarrow \text{Int} @-\}$$

To type `div x 5` we need $\{y : \text{Int} \mid y = 5\} \preceq \{y : \text{Int} \mid y \neq 0\}$

Refinement types, automated by SMT

$\{-@ \text{div} :: \text{Int} \rightarrow \{y:\text{Int} \mid y \neq 0\} \rightarrow \text{Int} @-\}$

To type `div x 5` we need $\{y : \text{Int} \mid y = 5\} \preceq \{y : \text{Int} \mid y \neq 0\}$

Subtyping on refinements is an implication, checked by the SMT solver:

$$\frac{\Gamma \wedge r_1 \Rightarrow r_2 \text{ is valid}}{\Gamma \vdash \{v : b \mid r_1\} \preceq \{v : b \mid r_2\}} \text{ (S-BASE)}$$

Refinement types, automated by SMT

$\{-@ \text{div} :: \text{Int} \rightarrow \{y:\text{Int} \mid y \neq 0\} \rightarrow \text{Int} @-\}$

To type `div x 5` we need $\{y : \text{Int} \mid y = 5\} \preceq \{y : \text{Int} \mid y \neq 0\}$

Subtyping on refinements is an implication, checked by the SMT solver:

$$\frac{\Gamma \wedge r_1 \Rightarrow r_2 \text{ is valid}}{\Gamma \vdash \{v : b \mid r_1\} \preceq \{v : b \mid r_2\}} \text{ (S-BASE)}$$

Validity is encoded as an unsatisfiability problem: $\Gamma \wedge r_1 \wedge \neg r_2$ is `unsat`

Refinement types, automated by SMT

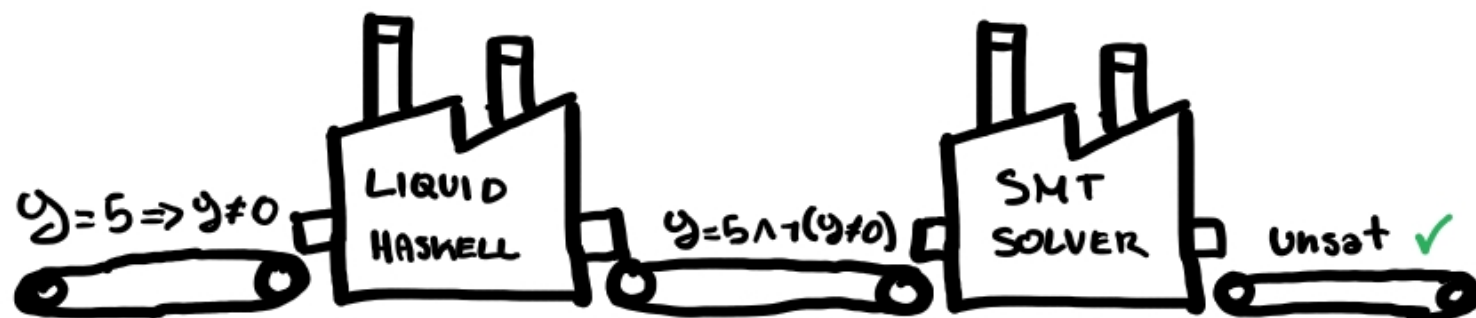
$\{-@ \text{div} :: \text{Int} \rightarrow \{y:\text{Int} \mid y \neq 0\} \rightarrow \text{Int} @-\}$

To type `div x 5` we need $\{y : \text{Int} \mid y = 5\} \preceq \{y : \text{Int} \mid y \neq 0\}$

Subtyping on refinements is an implication, checked by the SMT solver:

$$\frac{\Gamma \wedge r_1 \Rightarrow r_2 \text{ is valid}}{\Gamma \vdash \{v : b \mid r_1\} \preceq \{v : b \mid r_2\}} \text{ (S-BASE)}$$

Validity is encoded as an unsatisfiability problem: $\Gamma \wedge r_1 \wedge \neg r_2$ is `unsat`



SMT knows arithmetic, not higher-order functions

SMT solvers are good at dealing with arithmetic, equality, ...
but to verify functional programs,
we need to teach them about higher-order functions:
 β -reduction and η -expansion.

Running example: a correct compiler

Source: expressions

```
eval :: Expr → Int
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)
```

Target: a stack machine

```
{-@ exec :: o:OpCode
      → s:{Stack | operands o <= len s}
      → Stack @-}
exec (OpPush x) = push x
exec OpAdd = \case (x : y : xs) → (y + x) : xs
exec OpMul = \case (x : y : xs) → (y * x) : xs
```

The refinement makes the partial match in `exec` safe.

Programs indexed by their semantics

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

data Program where
{-@ PNil    :: Prop (Program id) @-}
    PNil    :: Program
{-@ PCons   :: op:OpCode
    → p:(Stack → {v:Stack | len v >= operands op}) → Prop (Program p)
    → Prop (Program (exec op . p)) @-}
    PCons   :: OpCode → (Stack → Stack) → Program → Program
```

A plain Haskell datatype: a list of opcodes.

Programs indexed by their semantics

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

data Program where
{-@ PNil    :: Prop (Program id) @-}
    PNil    :: Program
{-@ PCons   :: op:OpCode
    → p:(Stack → {v:Stack | len v >= operands op}) → Prop (Program p)
    → Prop (Program (exec op . p)) @-}
    PCons   :: OpCode → (Stack → Stack) → Program → Program
```

Think of `prop` as a function that attaches extra information to a value,
which we use to emulate **indexed types**.

Programs indexed by their semantics

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

data Program where
{-@ PNil    :: Prop (Program id) @-}
    PNil    :: Program
{-@ PCons   :: op:OpCode
    → p:(Stack → {v:Stack | len v >= operands op}) → Prop (Program p)
    → Prop (Program (exec op . p)) @-}
    PCons   :: OpCode → (Stack → Stack) → Program → Program
```

Read `Prop (Program f)` as “a stack program that **behaves like f**”.

The index is a **function** `Stack → Stack`.

Programs indexed by their semantics

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

data Program where
{-@ PNil    :: Prop (Program id) @-}
    PNil    :: Program
{-@ PCons   :: op:OpCode
    → p:(Stack → {v:Stack | len v >= operands op}) → Prop (Program p)
    → Prop (Program (exec op . p)) @-}
    PCons   :: OpCode → (Stack → Stack) → Program → Program
```

Correctness of the compiler is then a type:

```
{-@ compile :: e:Expr → Prop (Program (push $ eval e)) @-}
```

the compiled program behaves like pushing the value of e.

Composing programs

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

Composing programs

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

The PNil branch gives the verification condition:

$$\frac{\begin{array}{lcl} \text{prop } p_1 & = & \text{Program } s_1 \\ \wedge \text{ prop } p_2 & = & \text{Program } s_2 \\ \wedge \text{ prop } p_2 & = & \text{Program } id \end{array}}{\Rightarrow \text{prop } p_1 = \text{Program } (s_2 . s_1)}$$

(2)

(3a)

(3b)

(goal)

}

Composing programs

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

The PNil branch gives the verification condition:

$$\left. \begin{array}{lcl} \text{prop } p_1 & = & \text{Program } s_1 \\ \wedge \text{ prop } p_2 & = & \text{Program } s_2 \\ \wedge \text{ prop } p_2 & = & \text{Program } id \end{array} \right\} \begin{array}{l} (2) \\ (3a) \\ (3b) \end{array}$$

$$\Rightarrow \text{prop } p_1 = \text{Program } (s_2 . s_1) \quad (goal)$$

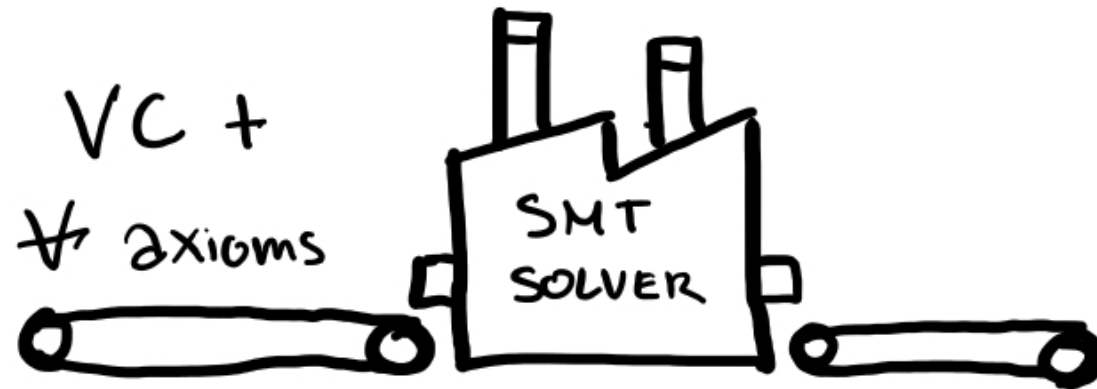
To the solver, s_1 , s_2 , id are just **constants**: it gets $s_2 = id$, but not $s_2 . s_1 = s_1$.

The traditional approach: axioms in the solver

Send the semantics to the solver as quantified axioms

$$\forall f\ g\ x. (f \cdot g)\ x = f\ (g\ x)$$

$$\forall x. id\ x = x$$

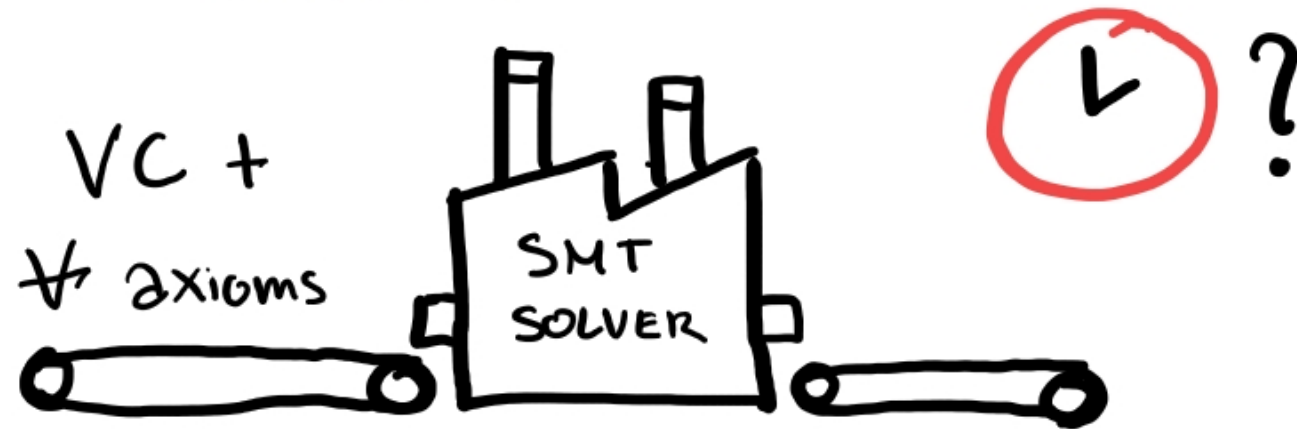


The traditional approach: axioms in the solver

Send the semantics to the solver as quantified axioms

$$\forall f\ g\ x. (f . g)\ x = f\ (g\ x)$$

$$\forall x. id\ x = x$$



$\forall$ is undecidable, so instantiation is heuristic: proofs are **slow** and **flaky**.

Just unfold the definitions!

Proof by Logical Evaluation (PLE) (Vazou et al., POPL'18) unfolds definitions itself:

$$(s_2 \cdot s_1) \times = s_2 (s_1 \times)$$

No $\forall$ reaches the solver: **decidable**, sound, terminating.

Not enough $\wedge$

Proof by Logical Evaluation (PLE) (Vazou et al., POPL'18) unfolds definitions itself:

$$(s_2 \cdot s_1) x = s_2 (s_1 x)$$

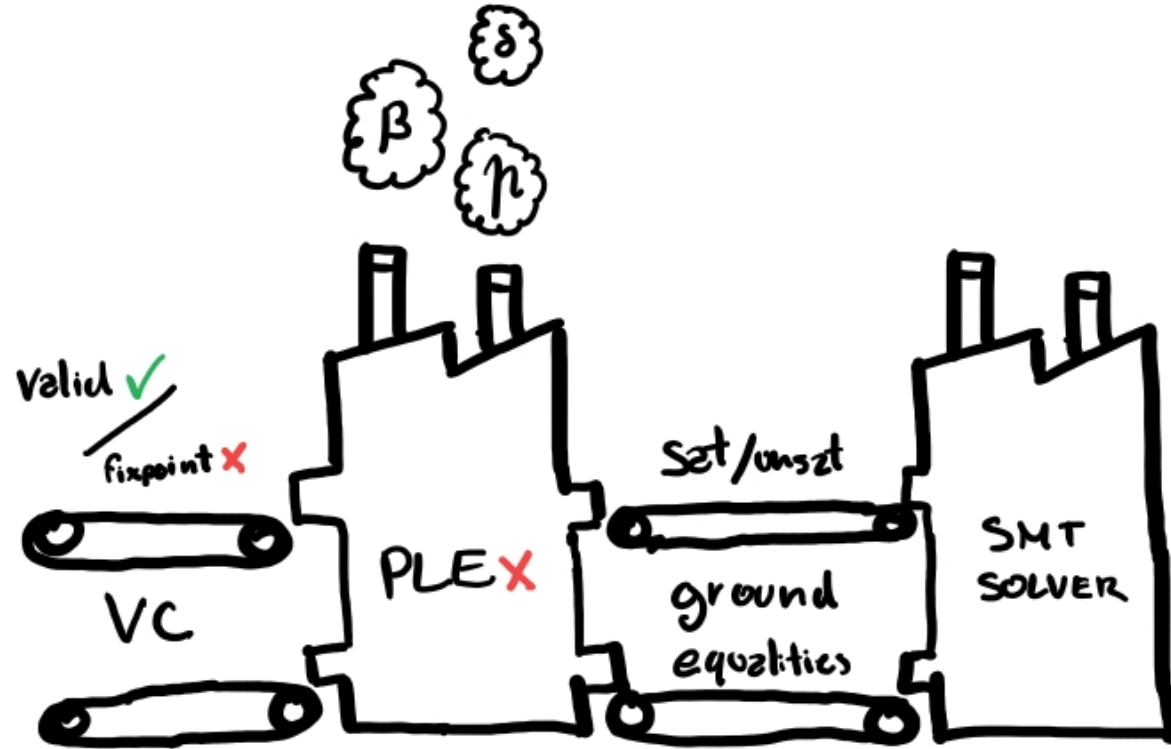
No $\forall$ reaches the solver: **decidable**, sound, terminating.

But on our VC it is stuck:

$s_2 \cdot s_1$ only **fully applied** functions unfold

$s_2 (s_1 x)$ $s_2 = id$ comes from pattern matching

$$P_{LEX} = P_{LE} + \beta\eta + \delta$$



The solver only ever sees **ground** equalities.

PLEX on our VC

	$\text{prop } p_1$	$=$	$\text{Program } s_1$	Equations with $=$ are added by Plex
			(2)	
$\wedge$	s_1	$=$	$\lambda x_4:\text{Stack}. s_1 \ x_4$	(4) η
$\wedge$	$s_2 . s_1$	$=$	$\lambda x_5:\text{Stack}. (s_2 . s_1) \ x_5$	(5) η

$$\Rightarrow \text{prop } p_1 = \text{Program } (s_2 . s_1) \quad (\text{goal})$$

η -expand partial applications to η -long normal form.

PLEX on our VC

	$\text{prop } p_1$	$=$	$\text{Program } s_1$	Equations with $=$ are added by Plex
				(2)
$\wedge$	s_1	$=$	$\lambda x_4:\text{Stack}. s_1 \ x_4$	(4) η
$\wedge$	$s_2 . s_1$	$=$	$\lambda x_5:\text{Stack}. (s_2 . s_1) \ x_5$	(5) η
$\wedge$	$(s_2 . s_1) \ x_5$	$=$	$s_2 (s_1 \ x_5)$	(6) $\text{unfold } (.)$

$$\Rightarrow \text{prop } p_1 = \text{Program } (s_2 . s_1) \quad (\text{goal})$$

Now $(s_2 . s_1) \ x_5$ is fully applied, so plain P_{LE} unfolding works.

PLEX on our VC

$$\begin{array}{llll}
 \text{prop } p_1 & = & \text{Program } s_1 & (2) \\
 \wedge \quad s_1 & = & \lambda x_4:\text{Stack}. s_1 \ x_4 & (4) \quad \eta \\
 \wedge \quad s_2 \cdot s_1 & = & \lambda x_5:\text{Stack}. (s_2 \cdot s_1) \ x_5 & (5) \quad \eta \\
 \wedge \quad (s_2 \cdot s_1) \ x_5 & = & s_2 (s_1 \ x_5) & (6) \quad \text{unfold } (.)
 \end{array}$$

Equations with $=$ are added by Plex

$$\frac{}{\Rightarrow \text{prop } p_1 = \text{Program } (s_2 \cdot s_1)} \quad (\text{goal})$$

s_2 is a variable: nothing to unfold. But T-CASE records what the match taught us:

$$\frac{\Gamma \vdash e : \{u : C \mid r_u\} \quad \Gamma, \overline{y : T_y}, x : \{u : C \mid r_u \wedge r_{xi}\} \vdash_{R, \Delta \cup \text{unif}(r_u \wedge r_{xi})} e_i : T}{\Gamma \vdash_{R, \Delta} \text{case } x = e \text{ of } \overline{d_i \ \overline{y} \rightarrow e_i} : T}$$

PNil branch: $\text{unif}(\text{prop } v = \text{Program } s_2 \wedge \text{prop } v = \text{Program } id) = \{s_2 \mapsto id\}$

PLEX on our VC

	$\text{prop } p_1$	$=$	$\text{Program } s_1$	Equations with $=$ are added by Plex
				(2)
$\wedge$	s_1	$=$	$\lambda x_4:\text{Stack}. s_1 \ x_4$	(4) η
$\wedge$	$s_2 . s_1$	$=$	$\lambda x_5:\text{Stack}. (s_2 . s_1) \ x_5$	(5) η
$\wedge$	$(s_2 . s_1) \ x_5$	$=$	$s_2 (s_1 \ x_5)$	(6) $\text{unfold } (.)$
$\wedge$	$s_2 (s_1 \ x_5)$	$=$	$id (s_1 \ x_5)$	(7) $\delta: s_2 \mapsto id$
$\Rightarrow$	$\text{prop } p_1$	$=$	$\text{Program } (s_2 . s_1)$	(<i>goal</i>)

$$\frac{(f \mapsto t_f) \in \Delta}{f \ \bar{t} = t_f \ \bar{t}} (\delta)$$

Pattern unification is unsound here: with $x \geq 10$, $f \ x = \text{true}$ has many solutions for f .

unif: first-order unification of variables and injective constructors, over the conjuncts of the refinement.

PLEX on our VC

	$\text{prop } p_1$	$=$	$\text{Program } s_1$	(2)	
$\wedge$	s_1	$=$	$\lambda x_4:\text{Stack}. s_1 \ x_4$	(4)	η
$\wedge$	$s_2 . s_1$	$=$	$\lambda x_5:\text{Stack}. (s_2 . s_1) \ x_5$	(5)	η
$\wedge$	$(s_2 . s_1) \ x_5$	$=$	$s_2 (s_1 \ x_5)$	(6)	unfold $(.)$
$\wedge$	$s_2 (s_1 \ x_5)$	$=$	$id (s_1 \ x_5)$	(7)	$\delta: s_2 \mapsto id$
$\wedge$	$id (s_1 \ x_5)$	$=$	$s_1 \ x_5$	(8)	unfold id
$\Rightarrow$	$\text{prop } p_1$	$=$	$\text{Program } (s_2 . s_1)$	(goal)	

Equations with $=$ are **added by Plex**

Unfold the now fully applied id .

PLEX on our VC

	$\text{prop } p_1$	$=$	$\text{Program } s_1$	(2)	
$\wedge$	s_1	$=$	$\lambda x_4:\text{Stack}. s_1 \ x_4$	(4)	η
$\wedge$	$s_2 . s_1$	$=$	$\lambda x_5:\text{Stack}. (s_2 . s_1) \ x_5$	(5)	η
$\wedge$	$(s_2 . s_1) \ x_5$	$=$	$s_2 (s_1 \ x_5)$	(6)	unfold $(.)$
$\wedge$	$s_2 (s_1 \ x_5)$	$=$	$id (s_1 \ x_5)$	(7)	$\delta: s_2 \mapsto id$
$\wedge$	$id (s_1 \ x_5)$	$=$	$s_1 \ x_5$	(8)	unfold id
$\Rightarrow$	$\text{prop } p_1$	$=$	$\text{Program } (s_2 . s_1)$	(goal)	

$$s_2 . s_1 \stackrel{(5)}{=} \lambda x_5:\text{Stack}. (s_2 . s_1) \ x_5 \quad \eta\text{-expand}$$

PLEX on our VC

	prop p_1	=	Program s_1	(2)	Equations with = are added by Plex
$\wedge$	s_1	=	$\lambda x_4:\text{Stack}. s_1 \ x_4$	(4)	η
$\wedge$	$s_2 . s_1$	=	$\lambda x_5:\text{Stack}. (s_2 . s_1) \ x_5$	(5)	η
$\wedge$	$(s_2 . s_1) \ x_5$	=	$s_2 (s_1 \ x_5)$	(6)	unfold $(.)$
$\wedge$	$s_2 (s_1 \ x_5)$	=	$id (s_1 \ x_5)$	(7)	$\delta: s_2 \mapsto id$
$\wedge$	$id (s_1 \ x_5)$	=	$s_1 \ x_5$	(8)	unfold id
$\Rightarrow$	prop p_1	=	Program $(s_2 . s_1)$	(goal)	

$$\begin{array}{lll}
 s_2 . s_1 & \stackrel{(5)}{=} & \lambda x_5:\text{Stack}. (s_2 . s_1) \ x_5 \quad \eta\text{-expand} \\
 & \stackrel{(6)}{=} & \lambda x_5:\text{Stack}. s_2 (s_1 \ x_5) \quad \text{unfold } (.)
 \end{array}$$

PLEX on our VC

	prop p_1	=	Program s_1	(2)	Equations with = are added by Plex
$\wedge$	s_1	=	$\lambda x_4:\text{Stack}. s_1 \ x_4$	(4)	η
$\wedge$	$s_2 . s_1$	=	$\lambda x_5:\text{Stack}. (s_2 . s_1) \ x_5$	(5)	η
$\wedge$	$(s_2 . s_1) \ x_5$	=	$s_2 (s_1 \ x_5)$	(6)	unfold $(.)$
$\wedge$	$s_2 (s_1 \ x_5)$	=	$id (s_1 \ x_5)$	(7)	$\delta: s_2 \mapsto id$
$\wedge$	$id (s_1 \ x_5)$	=	$s_1 \ x_5$	(8)	unfold id
$\Rightarrow$	prop p_1	=	Program $(s_2 . s_1)$	(goal)	

$$\begin{array}{lll}
 s_2 . s_1 & \stackrel{(5)}{=} & \lambda x_5:\text{Stack}. (s_2 . s_1) \ x_5 \quad \eta\text{-expand} \\
 & \stackrel{(6)}{=} & \lambda x_5:\text{Stack}. s_2 (s_1 \ x_5) \quad \text{unfold } (.) \\
 & \stackrel{(7)}{=} & \lambda x_5:\text{Stack}. id (s_1 \ x_5) \quad \delta, \text{ from the PNil match}
 \end{array}$$

PLEX on our VC

	prop p_1	=	Program s_1	(2)	Equations with = are added by Plex
$\wedge$	s_1	=	$\lambda x_4:\text{Stack}. s_1 \ x_4$	(4)	η
$\wedge$	$s_2 . s_1$	=	$\lambda x_5:\text{Stack}. (s_2 . s_1) \ x_5$	(5)	η
$\wedge$	$(s_2 . s_1) \ x_5$	=	$s_2 (s_1 \ x_5)$	(6)	unfold $(.)$
$\wedge$	$s_2 (s_1 \ x_5)$	=	$id (s_1 \ x_5)$	(7)	$\delta: s_2 \mapsto id$
$\wedge$	$id (s_1 \ x_5)$	=	$s_1 \ x_5$	(8)	unfold id
$\Rightarrow$	prop p_1	=	Program $(s_2 . s_1)$	(goal)	

$s_2 . s_1$	$\stackrel{(5)}{=}$	$\lambda x_5:\text{Stack}. (s_2 . s_1) \ x_5$	η -expand
	$\stackrel{(6)}{=}$	$\lambda x_5:\text{Stack}. s_2 (s_1 \ x_5)$	unfold $(.)$
	$\stackrel{(7)}{=}$	$\lambda x_5:\text{Stack}. id (s_1 \ x_5)$	δ , from the PNil match
	$\stackrel{(8)}{=}$	$\lambda x_5:\text{Stack}. s_1 \ x_5$	unfold id

PLEX on our VC

	prop p_1	=	Program s_1	(2)	Equations with = are added by Plex
$\wedge$	s_1	=	$\lambda x_4:\text{Stack}. s_1 \ x_4$	(4)	η
$\wedge$	$s_2 . s_1$	=	$\lambda x_5:\text{Stack}. (s_2 . s_1) \ x_5$	(5)	η
$\wedge$	$(s_2 . s_1) \ x_5$	=	$s_2 (s_1 \ x_5)$	(6)	unfold $(.)$
$\wedge$	$s_2 (s_1 \ x_5)$	=	$id (s_1 \ x_5)$	(7)	$\delta: s_2 \mapsto id$
$\wedge$	$id (s_1 \ x_5)$	=	$s_1 \ x_5$	(8)	unfold id
$\Rightarrow$	prop p_1	=	Program $(s_2 . s_1)$	(goal)	

$s_2 . s_1$	$\stackrel{(5)}{=}$	$\lambda x_5:\text{Stack}. (s_2 . s_1) \ x_5$	η -expand
	$\stackrel{(6)}{=}$	$\lambda x_5:\text{Stack}. s_2 (s_1 \ x_5)$	unfold $(.)$
	$\stackrel{(7)}{=}$	$\lambda x_5:\text{Stack}. id (s_1 \ x_5)$	δ , from the PNil match
	$\stackrel{(8)}{=}$	$\lambda x_5:\text{Stack}. s_1 \ x_5$	unfold id
	$\stackrel{(4)}{=}$	s_1	η , backwards

PLEX on our VC

	$\text{prop } p_1$	$=$	$\text{Program } s_1$	(2)	
$\wedge$	s_1	$=$	$\lambda x_4:\text{Stack}. s_1 \ x_4$	(4)	η
$\wedge$	$s_2 . s_1$	$=$	$\lambda x_5:\text{Stack}. (s_2 . s_1) \ x_5$	(5)	η
$\wedge$	$(s_2 . s_1) \ x_5$	$=$	$s_2 (s_1 \ x_5)$	(6)	unfold $(.)$
$\wedge$	$s_2 (s_1 \ x_5)$	$=$	$id (s_1 \ x_5)$	(7)	$\delta: s_2 \mapsto id$
$\wedge$	$id (s_1 \ x_5)$	$=$	$s_1 \ x_5$	(8)	unfold id
$\Rightarrow$	$\text{prop } p_1$	$=$	$\text{Program } (s_2 . s_1)$	(goal)	✓

$s_2 . s_1 \stackrel{(4-8)}{=} s_1$, so $\text{prop } p_1 \stackrel{(2)}{=} \text{Program } s_1 = \text{Program } (s_2 . s_1)$

Goal solved ✓

PLEX on our VC

			Equations with $=$ are added by Plex	
	$\text{prop } p_1$	$=$ Program s_1	(2)	
$\wedge$	s_1	$=$ $\lambda x_4:\text{Stack}. s_1 \ x_4$	(4)	η
$\wedge$	$s_2 . s_1$	$=$ $\lambda x_5:\text{Stack}. (s_2 . s_1) \ x_5$	(5)	η
$\wedge$	$(s_2 . s_1) \ x_5$	$=$ $s_2 (s_1 \ x_5)$	(6)	unfold $(.)$
$\wedge$	$s_2 (s_1 \ x_5)$	$=$ $id (s_1 \ x_5)$	(7)	$\delta: s_2 \mapsto id$
$\wedge$	$id (s_1 \ x_5)$	$=$ $s_1 \ x_5$	(8)	unfold id
$\Rightarrow$	$\text{prop } p_1$	$=$ Program $(s_2 . s_1)$	(goal)	✓

$s_2 . s_1 \stackrel{(4-8)}{=} s_1$, so $\text{prop } p_1 \stackrel{(2)}{=} \text{Program } s_1 = \text{Program } (s_2 . s_1)$

Goal solved ✓

Plain PLE was stuck: PLEX also needed η -expansion (4,5) and δ (7).
Every step is a **ground** equality, closed by congruence in the SMT solver.

Arithmetic for free: completing the compiler

```
{-@ compile :: e:Expr → Prop (Program (push $ eval e)) @-}
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) = ... -- as EAdd
compile (ENeg e)      = PCons OpMul ((push $ -1) . (push $ eval e))
                      (PCons (OpPush (-1)) (push $ eval e) (compile e))
```

Arithmetic for free: completing the compiler

```
{-@ compile :: e:Expr → Prop (Program (push $ eval e)) @-}
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) = ... -- as EAdd
compile (ENeg e)      = PCons OpMul ((push $ -1) . (push $ eval e))
                      (PCons (OpPush (-1)) (push $ eval e) (compile e))
```

PLEX handles the higher-order indices; the SMT solver still does the arithmetic.

ENeg: $-a = a * (-1)$ is proved automatically, no lemma needed.

Theorem (Soundness)

If $\text{PLEX}(\Gamma, R, \Delta, e)$ returns *true*, then $\forall \theta \in \llbracket \Gamma \rrbracket. \theta(e) \hookrightarrow^* \text{true}$.

Theorem (Soundness)

If $\text{PLEX}(\Gamma, R, \Delta, e)$ returns `true`, then $\forall \theta \in \llbracket \Gamma \rrbracket. \theta(e) \hookrightarrow^* \text{true}$.

Theorem (Unification)

$\text{unif}(xs, r) \subseteq \{x \mapsto e \mid r \Rightarrow x = e\}$, and `unif` terminates.

Theorem (Soundness)

If $\text{PLEX}(\Gamma, R, \Delta, e)$ returns `true`, then $\forall \theta \in \llbracket \Gamma \rrbracket. \theta(e) \hookrightarrow^* \text{true}$.

Theorem (Unification)

$\text{unif}(xs, r) \subseteq \{x \mapsto e \mid r \Rightarrow x = e\}$, and `unif` terminates.

Theorem (Termination)

If Γ is transparent and $\Gamma; R; \Delta \vdash e : \tau$, then $\text{PLEX}(\Gamma, R, \Delta, e)$ terminates.

Formal guarantees

Theorem (Soundness)

If $\text{PLEX}(\Gamma, R, \Delta, e)$ returns `true`, then $\forall \theta \in \llbracket \Gamma \rrbracket. \theta(e) \hookrightarrow^* \text{true}$.

Theorem (Unification)

$\text{unif}(xs, r) \subseteq \{x \mapsto e \mid r \Rightarrow x = e\}$, and `unif` terminates.

Theorem (Termination)

If Γ is transparent and $\Gamma; R; \Delta \vdash e : \tau$, then $\text{PLEX}(\Gamma, R, \Delta, e)$ terminates.

Refinement type checking with PLEX is **sound** and **decidable**.

Evaluation

Benchmark	δ	$\beta\eta$	Plex (Liquid Haskell)				Agda	
			Exec	Spec	Proof	Time (s)	LoC	Time (s)
Stack Compiler [running example]	✓	✓	53	19	0	3.56	69	6.73
STLC [Swierstra'23]	✓	✓	95	70	0	4.43	70	2.34
Compiler Exceptions [Pickard & Hutton'21]	×	✓	85	58	0	3.96	84	3.19
Map Fusion [GHC rewrite rules]	×	✓	22	12	6	1.90	27	1.17
List Monad	×	✓	28	16	11	2.24	59	1.35
Reader Monad	×	✓	13	9	4	2.11	29	1.08
State Monad	×	✓	12	8	3	2.15	31	1.21
Continuation Monad	×	✓	11	8	3	2.13	25	1.09
Total			319	200	27	22.58	394	17.20

0 / 8

verify with plain P_{LE}

$\approx$ **Agda**

proof effort

5%

slowdown on the LH test suite

Normalize **with** the SMT solver, in a loop:
the solver decides guards and goals, PLEX adds the equalities it cannot derive.

- ▶ $\beta\eta$ and δ sent as **ground equalities**
- ▶ **Sound** and **terminating**: decidable type checking
- ▶ Implemented in **Liquid Haskell**, 5% overhead



Paper

Thank you!